Structure-aware Abstraction Techniques for efficient Software Verification

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
Konstantin Britikov

under the supervision of
Prof. Dr. Natasha Sharygina

October 2026

Dissertation Committee

Prof. Dr. Patrick Eugster	Universita della Svizzera Italiana, Lugano, Switzerland
Prof. Dr. Carlo Furia	Universita della Svizzera Italiana, Lugano, Switzerland
Prof. Dr. Philipp Rümmer	Uppsala University, Uppsala, Sweden
Prof. Dr. Arie Gurfinkel	Waterloo University, Waterloo, Canada

Dissertation accepted on 26th October 2026

Research Advisor	PhD Program Director
Prof. Dr. Natasha Sharygina	**Prof. Dr. Walter Binder/ Prof. Dr. 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.

Konstantin Britikov
Lugano, 26th October 2026

The philosophers have only
interpreted the world in various
ways, the point is to change it.

Karl Marx

Abstract

Software systems are used in virtually every aspect of modern life. For safety-critical applications, such as automotive, avionics, or medical systems, it is essential to ensure that software does not malfunction or contain significant vulnerabilities. Various approaches exist to ensure the functional correctness of software; however, only *formal verification* can provide mathematical guarantees regarding the behavior of software systems with respect to their requirements. The advancements in SATisfiability (SAT) solving and Satisfiability Modulo Theories (SMT) have enabled the development of efficient automated safety verification techniques.

Software verification is undecidable in general, but intelligent heuristics and abstraction make it practical even for complex systems. State-of-the-art verification approaches can efficiently analyze industrial code, both proving that a system satisfies its requirements and detecting violations of its expected behavior *exhaustively* and *automatically*. These approaches check whether a program violates given requirements, either finding a concrete violating execution trace or proving that none exists.

One of the major challenges in software verification is the complexity of real-world programs. To make verification feasible, it is essential to abstract the program (reducing the size of the analyzed system) while preserving the details relevant to the verified properties. This balance between abstraction and concrete program representation is crucial for the efficient verification of complex programs. State-of-the-art techniques tend to abstract away the complex inner structure of the analyzed systems in various ways; however, software structure can be highly beneficial for verification. Structure allows the verification to be split into simpler subproblems corresponding to specific components.

This thesis focuses on improving the scalability of automated formal methods by finding a suitable balance between abstraction and concrete representation of programs, and by considering verification as an interconnected problem in its full complexity. To achieve this, the thesis investigates software safety verification at multiple layers. We identify three layers: *foundational*, *algorithmic*, and *application*. The foundational layer corresponds to SAT/SMT solving and Craig interpolation procedures. From the algorithmic perspective, the thesis focuses on the core symbolic logic rea-

soning techniques which form the basis of the computational procedures used both for the actual verification of programs and their abstract representation. Finally, the application layer investigates algorithms that use verification for other purposes, such as test generation. This multi-aspect approach allows us to focus on the important components of the problem while using some existing techniques as integral parts of general program verification. This means that the algorithmic solutions for verification employ the techniques from the foundational layer, and the application layer builds on safety verification algorithms. This thesis contributes to all three layers, as they are interconnected, and considering them holistically leads to more efficient software verification.

At the *foundational* layer of formal verification (SMT solving), the thesis proposes a new, flexible decision heuristic and investigates the influence of SMT solving heuristics on software verification and interpolant production. Building on the standard Conflict-Driven Clause Learning (CDCL) procedure, we constructed a new heuristic based on a combination of the Variable State Independent Decaying Sum (VSIDS) and Lookahead heuristics. This new procedure, called Picky CDCL, performs a more exhaustive check of the candidate literals than VSIDS, detecting more conflict clauses, while being more efficient than the classical Lookahead approach. Experimental evaluation showed that this new heuristic enabled the verification of previously unsolved safety problems, and demonstrated that the raw performance of an SMT solver does not necessarily lead to improved safety verification.

At the *algorithmic* layer, the thesis contributes two new approaches for the efficient verification of complex software systems and a technique for automated termination and nontermination analysis. Specifically, the thesis introduces a modular, structure-aware verification procedure that effectively handles loops and complex control flow in software. This structure-aware approach enables more efficient verification of programs with complex structure and multiple loops. The thesis additionally describes an extension of the proposed technique to handle software with nested loops. This extension relies on a compositional loop summarization that abstracts nested loops while preserving safety-critical information. Evaluation of the extension demonstrated the benefit of combining abstraction with software structural information. For termination analysis, we designed a new automated, safety-based approach that combines termination and nontermination analysis through their mutual guidance. It relies on safety verification to construct recurrent sets that witness nontermination and transition invariants that prove termination. This approach efficiently solves a significant number of benchmarks, proving termination and nontermination for instances that prior tools could not handle.

At the *application* layer, the thesis addresses a problem that builds on safety verification: automated test generation. For automated test generation, we implemented a

new tool, SolTG, which applies software verification techniques to generate test inputs: it checks the reachability of specific locations within a program and derives inputs that drive execution to reach them. Experimental evaluation of SolTG demonstrated its ability to generate test inputs and achieve high coverage for industrial software.

Acknowledgements

I would like to thank my advisor, Prof. Dr. Natasha Sharygina, for introducing me to the world of automated formal methods and for guiding me along the research path with patience and support. During my PhD, I had the opportunity to work with many talented and insightful scientists who helped and inspired me. In particular, I want to thank Prof. Grigory Fedyukovich for many conversations, fruitful discussions, and research insights; Dr. Martin Blicha for his guidance, help, and many discussions; and Dr. Antti Hyvärinen for introducing me to SMT solving and helping me navigate it. I am also grateful to my colleagues, blessed with a good sense of humour and always there when needed: Faezeh Labbaf, Dr. Tomáš Kolárik, Fabrizio Leopardi, Dr. Anna Becchi, Dr. Rodrigo Otoni, Dr. Sepideh Asadi, and Masoud Asadzadeh.

I would also like to thank the members of the committee for their time, valuable comments, and feedback. I gratefully acknowledge the financial support of the Swiss National Science Foundation.

Last but not least, I would like to thank the people who supported me, mentally and physically, on this long journey: my friends, my parents and family, and, most importantly, the love of my life, Lea.

Contents

Contents **xi**

List of Figures **xv**

List of Tables **xvii**

1 Introduction **1**
1.1 Automated Software Verification 1
1.2 Research Questions and Contributions 6
1.3 Thesis Structure 11

2 Background **13**
2.1 Satisfiability Modulo Theories Solving 13
2.1.1 Linear Integer Arithmetic 15
2.1.2 Craig Interpolation 16
2.1.3 OpenSMT Solver 17
2.2 Software Verification 18
2.2.1 Transition Systems 18
2.2.2 Transition Invariants 19
2.2.3 Safety Verification 19
2.2.4 Termination and Nontermination 20
2.2.5 Constrained Horn Clause Solving 22
2.2.6 Quantifier Elimination and Model-Based Projection 24
2.2.7 Golem CHC Solver 24
2.3 Smart Contracts and Their Verification 25

3 Structure-aware Modular Safety Verification of Software **27**
3.1 Preliminaries 28
3.2 Multiloop Analysis with TPA 31
3.2.1 Overview 31

3.2.2 The Core Algorithm . . . 32
3.2.3 Running Example . . . 35
3.2.4 Correctness . . . 36
3.2.5 Witness Production . . . 38
3.3 Evaluation of the Multiloop Verification . . . 40
3.4 The Multiloop Approach Extended to Nested Loops . . . 42
3.4.1 Nested Loops and Loop Summaries . . . 43
3.4.2 Extension of the Multiloop Approach to the Programs with Nested Loops . . . 45
3.4.3 Core Algorithm . . . 45
3.4.4 Refinement of Reached States . . . 49
3.4.5 Example . . . 52
3.4.6 Correctness . . . 53
3.4.7 Witness Production . . . 55
3.5 Evaluation of the Extended Multiloop Technique . . . 57
3.6 Limitations . . . 59
3.7 Related Work . . . 60
3.8 Future Work . . . 62
3.9 Conclusion . . . 63

4 Mutually Guided Termination and Nontermination Analysis 65
4.1 Preliminaries . . . 66
4.1.1 Proving Nontermination via Safety . . . 66
4.2 Termination Analysis via Interpolation-based Transition Invariant Generation . . . 69
4.2.1 Interpolation-based Generation of Well-Founded Transition Invariants . . . 70
4.2.2 Interpolation-based Termination Analysis . . . 74
4.2.3 Correctness . . . 76
4.3 Extended Interpolation-based Transition Invariant Generation for Proving and Disproving Termination . . . 77
4.3.1 Analysis of States for Which the Transition Invariant Is Not Valid 77
4.3.2 Overall Algorithm . . . 77
4.4 Implementation and Evaluation . . . 80
4.4.1 Implementation . . . 80
4.4.2 Evaluation . . . 81
4.5 Limitations . . . 82
4.6 Related Work . . . 83
4.7 Future Work . . . 84

4.8 Conclusion 84

5 Safety-based Automated Test Generation 87
5.1 SolTG: Test Case Generator for Solidity 88
5.1.1 Tool Overview 88
5.1.2 Support of Solidity Constructs 91
5.2 Test Case Generation from CHC Encoding 93
5.2.1 Solidity Smart Contracts to CHCs 93
5.2.2 Algorithmic Enumeration of Contract Behaviors 95
5.3 Evaluation 98
5.4 Limitations 99
5.5 Related Work 100
5.6 Future Work 101
5.7 Conclusion 102

6 Flexible SMT Solving for Safety Verification 103
6.1 Preliminaries 104
6.2 PickyCDCL 106
6.3 Related Work 109
6.4 Evaluation 110
6.4.1 Interpolation-based Algorithms 111
6.4.2 Proof Production 114
6.4.3 Overhead Evaluation 115
6.4.4 Lookahead Comparison 116
6.5 Limitations 117
6.6 Future Work 118
6.7 Conclusions 118

7 Conclusions 121

Figures

1.1 The three layers of software verification. 7

2.1 High-level architecture of OpenSMT. 17
2.2 C program and its symbolic encoding into a safety problem. 20
2.3 High-level architecture of Golem. 25

3.1 Multiloop example. 29
3.2 Cutpoint Graph for program in Figure 3.1. Transitions are labeled with constraints from *Fla*. 30
3.3 Algorithm execution flow for the example from Figure 3.2. 37
3.4 Comparison of performance of Golem-Multiloop with other tools: Z3-Spacer, Eldarica, Golem-Spacer and Golem-TPA. Plot on the left demonstrates the number of solved safe instances over time, plot on the right shows UNSAFE instances. 41
3.5 Nested loop example. 43
3.6 Cutpoint Graph for program in Figure 3.5. Transitions are labeled with constraints from *Fla*. 43
3.7 The algorithm execution flow for the example from Figure 3.6. 52
3.8 Comparison of CSRA with other engines in Golem. 58
3.9 Comparison of CSRA with state-of-the-art. 59

4.1 Execution flow of Safety-based Nontermination analysis. 67
4.2 Execution Flow of Interpolation-based Transition Invariant Generation . 70
4.3 Execution Flow of Interpolation-based Transition Invariant Generation (new components are placed in a gray block, lines 5,6,8). 74
4.4 Extended Interpolation-based Termination Analysis (new components are highlighted in dark orange and dark blue colors). 78
4.5 Comparison of ItpTig+ with KoAT, LoAT and T2 (runtime in seconds). 82

5.1 Architecture of SolTG. 89

5.2 High-level testing workflow with SolTG. 89
5.3 HTML coverage report. 90
5.4 SolTG performance for the benchmark set with different timeouts. . . . 98

6.1 A comparison of proof size for the VSIDS approach vs. PickyCDCL. (In the number of unique lemmas used.) 114
6.2 A comparison of the validation time for VSIDS-built proofs against the PickyCDCL-built proofs. (Time comparison in seconds.) 115
6.3 Overhead evaluation for OpenSMT with PickyCDCL-based vs. VSIDS-based solving for various depths of literal selection. (Time comparison in seconds.) 116
6.4 PickyCDCL-based vs. lookahead-based OpenSMT performance comparison for various depths of literal selection. (Time comparison in seconds.) 117

Tables

4.1 Comparison of ItpTig+ with SNA and ItpTig. 81
4.2 Comparison of ItpTig+ with state-of-the-art tools. 82

6.1 Golem results comparing various interpolation-based verification algorithms based on either VSIDS or PickyCDCL procedures of OpenSMT. . 112
6.2 Number of instances solved by virtual portfolios of Golem's TPA engine with different random seeds and SMT backends (3 engines per portfolio).113

Chapter 1

Introduction

Modern society relies on software systems present in every aspect of human life. In many domains software requires high reliability and resistance to failure, for example in medical or financial applications. Considerable effort and time are devoted to ensuring the correct behavior of such systems. One of the most common approaches to making software more reliable is *testing*. Testing can be effective, but even the most elaborate test suite cannot guarantee complete correctness: *it can only uncover errors, not prove their absence*. Moreover, test suites are written manually, demanding time and effort from developers. Additionally, human errors during the development of tests can lead to missed corner cases. These fundamental limitations pushed researchers toward the methods capable of **proving** correctness, leading to the creation and development of the field of software verification. Software verification techniques aim to provide formal guarantees of correctness, addressing the limitations of testing.

1.1 Automated Software Verification

Formal verification is a method that reasons about a system within a logical framework to prove that software satisfies certain requirements. It is a widely adopted method for ensuring the functional correctness of software systems, particularly for applications that require the complete absence of failures. Major organizations such as NASA, Amazon, Microsoft, and Meta actively employ formal methods to verify their software, and the adoption of these techniques continues to grow. Recent advancements in blockchain technology and the proliferation of smart contracts have further increased interest in formal methods [4, 153], as faults and errors in protocols and smart contracts can cost millions of dollars in losses. Another recent development is the emergence of verification applied to AI systems for the purpose of ensuring their reliability and intended behavior [129, 135, 176, 179].

The origins of formal reasoning about code can be traced back to Alan Turing's work on "Checking a Large Routine" [172]. Turing described the process of verification as follows: "*the programmer should make a number of definite assertions which can be checked individually, from which the correctness of the whole program easily follows*". Pioneers of computer science, such as Dijkstra, Hoare, and Floyd, sought to apply logic-based reasoning to prove program correctness [73, 87, 114]. One of the major contributions of the period is *Hoare logic*, which established a foundational paradigm for reasoning about programs that remains widely applied in formal verification today, especially in deductive and constraint-based verification. Hoare logic reasons about programs in terms of triples $\langle Pre, S, Post \rangle$, where a precondition *Pre* holds before execution, a statement S describes the execution, and a postcondition *Post* holds after execution of a statement. This allows formalizing both the execution of a program and the intended behavior of parts of software systems.

The early verification efforts relied heavily on manual, pen-and-paper approaches, with researchers reasoning about program assertions by hand. Applying such techniques to modern software systems would require an infeasible amount of effort, expertise, and time, as these systems are large and complex. To be truly accessible and widespread, verification must be *automated*, minimizing the additional burden placed on developers beyond the software development itself. State-of-the-art automated formal reasoning algorithms, developed by researchers in academia and industry, can be effectively applied even to complex, non-trivial systems such as distributed and concurrent software.

The most prominent method for automated software verification is *model checking* [80, 159], a Turing-Award-winning technology widely adopted by the hardware and software industry. The main idea of model checking is to construct a formal model [164] of the system. Reasoning about this model allows verifying or disproving the system's properties, using algorithms that ensure an exhaustive traversal of the system. The model can be represented in various ways, such as finite-state automata or sets of first-order logic formulae. A key advantage of model checking is its automation, as it does not require any manual effort other than defining the system property to be verified. The main obstacle for model checking is *state-space explosion*: the number of states grows exponentially with the size of the system, and some systems have infinitely many states, which makes the problem undecidable in general. Modern techniques nonetheless reason about such systems efficiently, using abstraction to keep the analyzed state space tractable and applying advanced algorithms to explore it.

In the context of model checking, one of the most important tasks is to check the system's "safety". A software system is called safe when "bad things never happen" [131]. When a program is not safe, there exists a finite path through the program reaching a violation that serves as a counterexample. Safety is often reduced to reachability checks,

establishing whether a "violation state" (representing the violation of the safety property) can be reached within a system. The majority of modern model checkers can produce either a counterexample that demonstrates a violation of the system's safety (an error trace) or a correctness witness that guarantees the absence of violations (a system invariant). Safety checks can also be used for other purposes, such as automated test generation (to reach specific locations within a program) and nontermination analysis (to show that a certain state can be reached infinitely often), among others.

One of the most significant breakthroughs in model checking was the idea of using a symbolic representation of the system, which made model checking efficient. The first symbolic encodings used Binary Decision Diagrams (BDDs) [54], and later propositional logic formulae that encode the model as a satisfiability (SAT) query [31]. This development allowed model checking to benefit from advances in SAT solving, a technique that forms the core of the constraint-solving field. The emergence of Satisfiability Modulo Theories (SMT) solvers allowed model checking to be based on first-order logic rather than pure propositional logic [7]. This improved both the performance of model checkers and the fidelity of models to the original systems, especially for software.

The introduction of symbolic SAT-based system modeling coincided with the arrival of Bounded Model Checking (BMC), a very efficient technique for counterexample detection. BMC is a bounded algorithm: it checks for the existence of a counterexample to safety by searching for a path of a certain length within a system that violates the safety property. The bound is increased iteratively until a counterexample is found or the whole state space is explored, proving that no state of the system violates the safety property. Even though it was initially proposed for hardware systems, it was eventually extended to software [64]. However, BMC has an important drawback: it cannot prove the safety of an infinite-state system, and it needs to enumerate all reachable states to prove the safety of finite systems. To address this limitation of BMC, a different technique called *k-induction* [165] was introduced. It still used BMC as a base for the counterexample search but, as an additional step, the algorithm attempted to construct an inductive proof of safety. The next step in the verification of unbounded systems was the introduction of Interpolation-Based Model Checking (IMC) [142]. Building on BMC's framework, this method relies heavily on the concept of Craig interpolation [67] to compute an overapproximation of the reachable states. Interpolation allows constructing an invariant of the system, and if the invariant does not violate the safety property, the system is safe. IMC has made the concept of Craig interpolation (interpolation from now on) very popular in the formal methods community, and it is now used in a wide variety of state-of-the-art model checking techniques.

Interpolation is closely related to the notion of abstraction in formal verification. The abstraction-based approach implies dropping the details of the system that are not relevant to the verified property. It can significantly reduce the number of considered

states, addressing the potential state-space explosion problem and simplifying the original system. The most commonly used abstraction techniques in software verification rely on the concept of overapproximation. When an overapproximated system is safe, the original system is also safe. However, overapproximation can introduce spurious behaviors not present in the original system. In this case, a counterexample in the abstraction does not mean that a counterexample exists in the original system, and the abstraction needs to be *refined*. One of the earliest techniques for abstraction used in program verification is *predicate abstraction*, which uses a set of predicates over the program variables to construct abstract states [12, 102]. The construction of such abstractions was fully automatic; however, no mechanism for automatically refining the produced abstraction was proposed. An efficient automated refinement technique was introduced with the Counterexample-Guided Abstraction Refinement (CEGAR) [63]. CEGAR automatically detects spurious counterexamples and refines the abstraction to eliminate them. A further advancement over CEGAR was achieved by applying *Lazy Abstraction* [110], which performs refinement on demand, without re-analyzing the refined system from scratch. Interpolants were also incorporated into Lazy Abstraction through the *Lazy Abstraction with Interpolants* approach [145], enabling automatic discovery of relevant predicates.

The early techniques described above were based on the advancements in SAT solving. A further advancement came as SMT solving grew increasingly efficient [95]. Efficient SMT solving procedures made it possible to reason about more complex concepts, such as arithmetic and data structures. Overall, this led to more efficient verification of software systems, as it enabled modeling systems and encoding safety properties closer to their original representation. The use of SMT as the underlying computational engine of verification advanced the field of software verification immensely, allowing efficient verification of real-world software systems. Many existing techniques, such as BMC and k-induction [74], were extended to SMT-based software verification. Predicate abstraction [130], CEGAR [26], and interpolation-based model checking algorithms [3, 143] were also adapted for SMT-based verification.

Another significant development in the field of symbolic model checking is the introduction of the IC3 [42, 43] algorithm. It was initially proposed for hardware model checking and was later generalized into Property-Directed Reachability (PDR) [115], which extended it to the SMT setting. Unlike the previously mentioned approaches, PDR works by maintaining a sequence of overapproximations of the reachable states at each time step, incrementally strengthening them into an inductive invariant. This approach has been heavily studied in recent years, with various modifications proposed. One notable evolution of PDR is the *Spacer* algorithm [124], which introduced underapproximating summaries and the use of Model-Based Projection (MBP). MBP allows the efficient computation of predecessor states by underapproximating quanti-

fier elimination. Property-directed k-induction [119] was another development in the field, replacing PDR's standard single-step induction with k-inductive reasoning.

Among the recent advances in model checking that are complementary to the traditional techniques is the Transition Power Abstraction (TPA) [34, 35], which proves safety based on the overapproximation of the transition relation and the construction of a transition invariant. TPA overapproximates up to 2^n transitions per iteration, which allows it to perform deep reachability checks very quickly. Another interesting development is the use of underapproximation for software verification. Underapproximation is particularly effective for the detection of counterexamples, as counterexamples found in the underapproximated system also exist in the original system. If the underapproximation is safe, it does not mean that the overall system is safe, as it can miss some behaviors of the original system. In particular, the recently introduced Accelerated Bounded Model Checking (ABMC) [91] uses underapproximation to abstract away loops in software, allowing it to quickly traverse deep loops. Both TPA and ABMC explore the idea of abstraction over the transition relation instead of the state space. Another recent approach of interest is Infinite State Model Checking by Learning Transitive Relations (TRL) [93], which extends the analyzed system with transitive relations, overapproximating transition relations to prove safety. The algorithm introduces transitive relations until a constant number of steps becomes sufficient to cover all reachable states, after which it proves safety by checking that no error state is reachable within this constant number of transitions.

Most state-of-the-art model checking techniques listed above rely on SAT/SMT solving backends. These techniques encode safety verification problems as SAT/SMT queries and delegate them to underlying SAT/SMT solvers such as Z3 [72], CVC5 [13], OpenSMT [117], Yices 2 [78], and many others [60, 61], leveraging highly efficient solvers as black-box engines.

SMT solvers are optimized for solving logical queries, but the most performant solvers are not necessarily the most effective for software verification. Many of the aforementioned model checking techniques depend on SMT solving artefacts, such as Craig interpolants and variable assignments. Both the characteristics of these artefacts and the time it takes to produce them have a substantial impact on the model checking process. Moreover, direct encoding of software into SMT queries is typically inefficient; the majority of modern model checkers operate on some form of intermediate representation between the SAT/SMT encoding and the actual software. This intermediate representation and the corresponding technique for producing SMT queries significantly influence the verification process.

One prominent intermediate representation of particular interest in the context of software model checking is Constrained Horn Clauses (CHC) [103]. This framework provides a purely logical language (a subset of SMT) that preserves the structure of a

system's control flow. Depending on the encoding, it is possible to directly map parts of the CHC model to their exact counterparts in the software (down to specific lines of code). Due to the universality of logical representation, CHCs can be used to model not only software but also distributed protocols, concurrency, recursion, hardware systems, and many other domains. This representation is employed by numerous state-of-the-art model checkers, such as Korn [81], SeaHorn [106], RustHorn [140], Tricera [82], SolCMC [4], and JayHorn [161]. The CHC representation allows the problems of modeling and verification to be considered separately. Tools that solve CHC systems, known as *CHC solvers* — for example, Z3, Golem [33], Eldarica [116], and Theta [10] — can verify the safety of a program written in any programming language, provided the program can be translated into CHCs.

One of the main challenges in software verification is scalability, in particular, state-of-the-art tools often struggle with complex software systems that contain multiple loops and divergent control flow. Since software verification is undecidable in general, verification algorithms and their abstraction techniques must restrict the analysis to decidable fragments of the problem, and must execute verification procedures efficiently. Although many model checking approaches target this problem, they do not fully exploit software structural information, and are difficult to combine within a single verification framework. Model checkers based on IC3/Spacer tend to be inefficient for the analysis of software with deep loops. In contrast, techniques like TPA, ABMC, and TRL attempt to abstract away loop iterations, enabling faster system traversal. However, these techniques can struggle with complex software systems whose structural information could be highly beneficial for verifying the system's properties. These open challenges motivate the contributions presented in this thesis.

1.2 Research Questions and Contributions

The thesis focuses on the challenges of efficient automated software verification. Particularly, it aims at improving the scalability of software verification techniques for software with complex control flow (such as a presence of multiple loops and nested loops), at strengthening termination and nontermination analysis through mutual guidance of safety-based techniques, and at adapting safety verification methods for efficiently solving other problems, such as automated and exhaustive test generation. We exclude the problem of the system modeling from the scope of the thesis, relying on the existing CHC representations and translations from programming languages into CHCs. Overall, the thesis investigates the challenges of software verification which arise at different levels such as the *Foundational layer* — SAT and SMT solving, the backbone of modern model checking techniques, heavily used during verification, yet usually

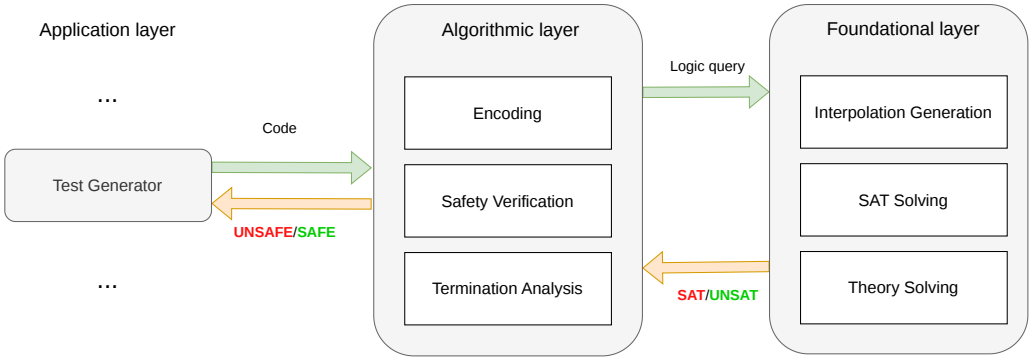


Figure 1.1. The three layers of software verification.

treated as a black box by most verification tools. The efficiency of these techniques is crucial for safety verification. The *Algorithmic layer* comprises different techniques and approaches for model checking. These techniques are a key element in detecting counterexamples to safety, in proving a system safe, and in analyzing systems for termination and nontermination. The algorithmic layer uses the foundational layer for many purposes, such as reachability checks and the generation of interpolants. The *Application layer* refers to the applications of model checking for specific purposes, for example, automated test generation. The application layer uses safety checks to reason about the system. Figure 1.1 illustrates the three layers investigated in this thesis and gives a high-level view of their interaction. The thesis hypothesis is that verification can be improved considerably when considered as an interconnected, integrated problem spanning all of the aforementioned layers.

The thesis investigates the following research questions.

Question 1: How can complex multiloop software with deep loops be verified efficiently? (Algorithmic layer, Chapter 3) One of the central contributions of this thesis addresses the challenge of verifying software with multiple loops and a large number of loop iterations [47]. Programs with interconnected loops that can iterate hundreds or thousands of times are complex to analyze. State-of-the-art model checking techniques such as IC3/Spacer tend to be inefficient when confronted with deep loops, while approaches such as TPA, designed specifically for deep loop exploration, are limited to single-loop programs. It is possible to use TPA to verify such systems by encoding the complex software structure as a single loop; however, such an encoding discards the structural information of the program and can significantly slow down verification, or even make it infeasible.

This thesis presents a new approach, *Multiloop-TPA*, which applies TPA modularly and incrementally across the individual loops of a program. The program is repre-

sented as a *Cutpoint Graph* (CPG), a compact structure in which each node corresponds to a loop head and edges encode loop-free paths between loop heads as symbolic formulae over program variables. The algorithm performs a forward reachability analysis over the CPG, gradually building a path from the program entry toward the error state. At each step it alternates between two phases: a *simple check*, which uses a single SMT query to test whether a single transition between two loop heads is feasible, and a *loop check*, which invokes TPA to decide whether a given loop can be traversed in an arbitrary number of iterations. When the current path cannot be extended, the algorithm backtracks and learns *blocking lemmas* — symbolic formulae characterizing states from which the error is unreachable — which are propagated both forward and backward along the explored path, eliminating redundant work during subsequent exploration attempts. Transition invariants produced by TPA for a loop are reused across multiple reachability queries for that loop, enabling incremental learning that significantly reduces the cost of its repeated analysis.

This approach has been implemented inside the Golem CHC solver and evaluated on dedicated benchmark suites. For multiloop programs, the proposed approach solved significantly more instances than the next-best competitors, on both safe and unsafe benchmarks.

Question 2: How can software with nested loops be verified efficiently? (Algorithmic layer, Chapter 3) Extending the research on the previous question, multiloop systems become even more difficult to verify when the program contains nested loops. Nested loops complicate safety verification, as they potentially require unrolling the same loop multiple times (a nested loop can be reached repeatedly within the outer loop). Very few state-of-the-art techniques include optimizations specifically targeting nested loops. This thesis specifically addresses multiloop systems with nested loops [48].

The core idea is to replace loops with overapproximating loop summaries that are refined on demand. As in the analysis of multiloop systems, the structural information about the software is used to compose nested-loop summaries. The loop abstraction is itself nested: the summary of an outer loop is constructed using the overapproximation of its inner loops. The refinement and construction of loop summaries are made possible by the TPA, in particular its ability to produce transition invariants that serve as loop summaries. Additionally, TPA allows the exact counterexample traces to be constructed efficiently, since it can traverse loops much faster than BMC-based approaches. Summaries are updated lazily: only when the algorithm detects a counterexample and the concrete counterexample construction reveals that it is spurious. To analyze the system, the algorithm constructs traces through the system without entering the nested loops, overapproximating loop execution via the loop summary.

When such overapproximated traces reach a counterexample, the algorithm unrolls the loops to construct the exact path through the system.

This technique was implemented inside the GOLEM CHC solver. It achieved the best performance among the state-of-the-art tools (LoAT, Spacer, Eldarica), solving the largest number of nested-loop instances.

Question 3: Can mutual guidance of termination and nontermination analysis techniques strengthen safety-based termination analysis? (Algorithmic layer, Chapter 4) Termination and nontermination analysis are one of the most difficult tasks in formal verification. Several approaches have reduced both termination [30, 65] and nontermination [59] to safety verification. However, state-of-the-art approaches predominantly do not rely on safety checks and employ separate techniques for termination and nontermination. For nontermination, most techniques rely on detecting a *recurrent set* — a set of states in which every state can reach at least one other state in the set, and which contains at least one initial state [89, 90, 132]. For termination, the majority of techniques generate a ranking function over the transition relation that proves the system eventually terminates [22, 163, 174]. An alternative technique is the construction of a *well-founded transition invariant*, the existence of which proves termination [157].

This research proposes combining termination and nontermination techniques, arguing that each can guide the other during the analysis. The new technique executes a safety-based nontermination routine, attempting to detect a system invariant that is simultaneously a recurrent set. Additionally, it uses the artefacts of safety calls — detected counterexample traces — to approximate those traces and construct a candidate well-founded transition invariant. When the candidate formula is a transition invariant over all reachable states, the algorithm concludes that the transition system is terminating. Otherwise, if the safety invariant constitutes a recurrent set, the algorithm concludes nontermination.

The proposed approach was implemented as the termination analysis engine inside of the GOLEM CHC solver. The evaluation demonstrated that the combination of termination and nontermination techniques is mutually beneficial. The comparison with state-of-the-art tools (LoAT, KoAT [137], T2 [51]) showed that our approach is competitive and orthogonal, solving a significant number of unique benchmarks, including instances never solved in the termination competition.

Question 4: How can tests for Solidity smart contracts be generated automatically using safety checking? (Application layer, Chapter 5) Safety verification has multiple applications beyond the direct check for safety, such as automated test gener-

ation. The main drawback of testing is that it relies on a significant amount of manual effort, requiring developers to identify corner cases and cover all possible execution behaviors of the system. Automated techniques such as fuzz testing address this, but they do not guarantee full test coverage. Unlike classical test generation techniques, formal verification provides a more guided automated approach. Safety-based techniques enumerate all locations in a program and check the reachability of each location using safety verification.

To address this question, this thesis proposes a solution, implemented in a new tool called SolTG [50], the first safety-based automated unit test generator for the Solidity programming language. The tool translates Solidity smart contracts into a CHC representation and uses this representation to reach particular locations in the contract code by executing safety checks. When a safety check detects a counterexample, SolTG generates the corresponding test case. As output, SolTG produces executable unit tests against which the system can be checked for coverage. The algorithm keeps generating tests until all possible branches are covered or a timeout is reached. SolTG also supports Solidity-specific features such as gas, Ethereum transfers, and blockchain addresses.

The evaluation conducted with SolTG demonstrated that the tool efficiently generates extensive test coverage, detecting corner cases in Solidity code and producing tests that exercise them.

Question 5: Can an SMT decision heuristic that performs a more thorough search yield better interpolants for safety verification? (Foundational layer, Chapter 6) The decision heuristic is the basis of any SMT solving approach. The order in which literals are selected for assignment is critically important and can significantly influence the solving process, to the extent that a different heuristic choice can allow the solver to decide an instance that previously timed out. The heuristic used in most SMT solvers is Variable State Independent Decaying Sum (VSIDS) [149], employed within the Conflict-Driven Clause Learning (CDCL) framework [167]. VSIDS ranks literals by the number of times they have participated in conflicts during solving, and selects the highest-ranked literal for the next assignment decision.

A contribution of this thesis [49] is a new heuristic called PickyCDCL, which allows greater flexibility in literal selection. It maintains a candidate set of δ top-ranked VSIDS literals and applies a lookahead procedure to select the one that leads to the greatest number of subsequent unit propagations. This new technique was implemented and evaluated on unsatisfiability proof generation, interpolant generation, model checking, and overall SMT solving performance.

The evaluation shows that a safety verification approach that used PickyCDCL in the SMT solver, instead of the traditional VSIDS heuristic, was able to uniquely verify

a significant number of instances from the CHC competition, in both Linear Integer and Linear Real Arithmetic. Furthermore, the more deliberate literal selection led to a substantial reduction in proof validation time, making validation 1.7 times faster than for proofs produced with standard CDCL. In terms of raw solving performance, PICKYCDCL introduced only a modest overhead, remaining competitive with the original approach.

Overall, this line of research demonstrates the direct and non-trivial connection between the SMT solving heuristic and the downstream model checking techniques that rely on it. A targeted change in literal selection had a substantial impact on interpolant quality, which in turn significantly influenced the model checking results.

1.3 Thesis Structure

Chapter 2 defines the background and notation used throughout the thesis, covering SMT, Craig interpolation, transition systems, the safety problem, termination and nontermination, Constrained Horn Clauses, and smart-contract verification. The subsequent chapters present our contributions. Chapter 3 introduces approaches for verifying complex multiloop and nested-loop software systems, and Chapter 4 presents a novel safety verification-based technique for termination and nontermination analysis — together presenting our contributions to the algorithmic layer. Chapter 5 presents a safety-based automated test generator for Solidity smart contracts, our contribution to the application layer. Chapter 6 describes the PICKYCDCL heuristic and investigates its influence on software verification, presenting our contribution to the foundational layer. The thesis concludes with Chapter 7, which summarizes our findings and discusses directions for future work. The chapters are ordered by the significance of their contribution rather than by layer: the algorithmic contributions to safety and termination analysis (Chapters 3 and 4) form the core of the thesis, and they are followed by the application of safety checking to test generation (Chapter 5) and by the foundational contribution to SMT solving (Chapter 6).

Chapter 2

Background

This chapter introduces the background necessary for the description of the contributions, developed in the course of the PhD. The research mainly concerns automated software verification, specifically verifying safety and termination/nontermination properties of software systems within the framework of symbolic model checking. The chapter also gives an overview of termination and nontermination analysis and of smart-contract verification, the application domain of Chapter 5. Programs are modeled with decidable fragments of first-order logic using the SMT and CHC-based representations.

2.1 Satisfiability Modulo Theories Solving

The Satisfiability (SAT) problem [32] is one of the fundamental problems in computer science. The problem is defined as follows: ”Given a boolean formula, does there exist an assignment of boolean variables that evaluates the formula to true ($\top$)?”. However, even though the formulation is simple, solving SAT is an NP-complete problem. Despite the complexity of the problem, there exists an extensive body of research on SAT solving, which allows one to efficiently solve many instances of the problem in practice. A core reason for the popularity of SAT solving is that many problems can be reduced to SAT, especially in the context of hardware and software verification. SAT solvers turned out to be very efficient in practice for verification purposes, and are widely used in industry [79, 175].

However, many problems cannot be reduced to SAT, and require more expressive formalisms. Software verification often requires reasoning about data structures, arithmetic, strings, and other complex domains. Researchers addressed this issue by combining SAT solving with decision procedures for various theories in first-order logic. This led to the development of Satisfiability Modulo Theories (SMT) solving [20], which addresses formulae in specific fragments of first-order logic, such as linear inte-

ger arithmetic, arrays, bit-vectors, and others. The tools that solve SMT problems are called SMT solvers. The SMT solving research domain has been very active in the last two decades, and has been standardized by the international SMT-LIB initiative [16]. This initiative defines the syntax and semantics of SMT formulae and the standard set of theories, maintains a set of SMT benchmarks, organizes yearly SMT workshops, and hosts the annual SMT competition [18], which is the main venue for evaluating and comparing SMT solvers. SMT-LIB also maintains a standard input format for SMT solvers ".smt2" [17], which is widely used in the research community and in industry.

Syntax We work in some fragments of first-order logic defined by a signature Θ, which consists of a set of function symbols, predicate symbols, and constant symbols. The logic also uses logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ (conjunction, disjunction, negation, implication), and quantifiers $\forall$, $\exists$. A *term* is a variable or a function symbol applied to terms. An *atom* is a predicate symbol applied to terms or a boolean term. We consider *multi-sorted* logical formulae, where variables are associated with a specific sort. The most common sort is *boolean*, which serves as the basis for the syntax of SMT formulae. Similarly, functions and predicates are associated with specific sorts for their arguments. Functions additionally have a sort for their return value, while for predicates it is implicitly boolean. The equality symbol $=$ is part of every theory's signature, and is interpreted as the equality relation on the domain of each sort. A *literal* is an atom or its negation. A *clause* is a disjunction of literals. There exist two common normal forms for formulae: Conjunctive Normal Form (CNF), which is a conjunction of clauses, and Disjunctive Normal Form (DNF), which is a disjunction of conjunctions of literals. SMT solvers internally convert input formulae to CNF for the underlying SAT engine. In some specific cases formulae may also be processed into DNF. This thesis is mostly focused on quantifier-free formulae (i.e., formulae containing no quantifiers, whose satisfiability amounts to finding a satisfying variable assignment). Quantified formulae appear in Chapter 4, however, the quantified formulae are processed into the quantifier-free ones using quantifier elimination.

Semantics SMT solving decides the satisfiability (SAT) of a given formula, producing a model if formula is satisfiable. A model M consists of a non-empty set called the universe of the model and an interpretation for the symbols, mapping function symbols to functions and predicate symbols to relations over the universe. A formula ϕ is satisfiable if there exists a model M such that ϕ evaluates to true: $M \models \phi$. The satisfiability in SMT is defined with respect to a specific theory T, which specifies the interpretation of the symbols. Different theories include, for example, Linear Integer Arithmetic (LIA), Linear Real Arithmetic (LRA), Arrays (AX), Uninterpreted Functions

(UF), and many others. A model specifies the interpretation of the formula's free variables in particular. A formula is T-valid if it evaluates to true in all models of T, and is T-satisfiable if it evaluates to true in at least one model.

If the solver decides that the formula is satisfiable, it returns the result (SAT) and a model M that provides a satisfying variable assignment. Otherwise, the solver returns UNSAT, and may additionally provide a proof of unsatisfiability or an interpolant for some part of the formula, described in more detail below. This thesis is particularly interested in the LIA theory, which is very common in software verification.

2.1.1 Linear Integer Arithmetic

Definition 1 (Linear Integer Arithmetic)**.** *The syntax of linear integer arithmetic (LIA) is defined as follows (it is also combined with the standard boolean variables and connectives)*[1]*:*

$$\begin{aligned}
\phi &\triangleq t_b \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid e_1 \leq e_2 \mid e_1 = e_2 \mid b_1 = b_2 \mid \neg\phi && \text{(LIA Formula)}\\
e &\triangleq t_i \mid e_1 + e_2 \mid -e && \text{(Arithmetic Expression)}\\
t_i &\triangleq 0 \mid 1 \mid \ldots \mid x_0 \mid x_1 \mid \ldots && \text{(Integer Terms)}\\
t_b &\triangleq \top \mid \bot \mid b_0 \mid b_1 \mid \ldots && \text{(Boolean Terms)}
\end{aligned}$$

Where $x_0, \ldots, x_n$ are integer variables and $b_0, \ldots, b_1$ are boolean variables.

We consider linear arithmetic over the domain of integers. In the syntax of Definition 1, the terms are defined as integer or boolean constants and variables, and atoms are comparisons over the terms that evaluate to boolean values. LIA corresponds to the theory of Presburger arithmetic, with the usual interpretation of the symbols. For example, multiplication of a variable by a constant $c \cdot v$ is interpreted as $v + v + \ldots + v$ (c times), and the minus operation $x_0 - x_1$ can be described as a sum $x_0 + -1 \cdot x_1$.

Example 1. *Formula $x > 12 \wedge y < 5 \wedge x + y > 13$ is satisfiable in LIA, with a possible model ($x = 21$, $y = 0$).*

The quantifier-free fragment of LIA is decidable. Usually, LIA solving is built on top of the Conflict-Driven Clause Learning (CDCL) SAT-solving architecture within CDCL(T) framework. As a basis, it uses decision procedures for Linear Real Arithmetic, which is a simpler theory than LIA. The SMT solver checks if the formula is satisfiable over the reals. If the solution assigns non-integer values to integer variables, new constraints are added to exclude the non-integer solution.

[1] $\triangleq$ is used through thesis to give an interpretation to a specific predicate.

2.1.2 Craig Interpolation

Given two formulae A and B where $A \wedge B$ is unsatisfiable, a Craig interpolant for A and B is a formula I, where: (1) $A \rightarrow I$; (2) $I \wedge B$ is unsatisfiable; and (3) I only contains the symbols common to both A and B. Craig interpolants are widely used to construct overapproximations of state space. This thesis uses $\textsc{Itp}(A, B)$ to describe the computation of a Craig interpolant.

Craig interpolants can be produced as a byproduct of SMT solving for theories like LIA, LRA, UF, AX [46, 62, 122]. Interpolation is widely used in formal methods, particularly in the context of model checking, where it serves as an efficient tool for overapproximation of reachable states and the transition relation, particularly in the construction of invariants. In particular, it is used in techniques such as IMC, LAWI, TPA and many others. Generally, the interpolant is not unique, and the choice of the interpolant can have a significant impact on the verification process. Interpolants can be computed with a variety of methods that differ in the logical strength, size, and structure of the interpolants they produce, and these differences can guide the verification process in different ways. At the propositional level, an interpolant is extracted from a resolution proof of unsatisfiability, for instance with McMillan's system [144] or Pudlák's symmetric system [158]; the Labeled Interpolation Systems generalise both into a parametric family whose members are ordered by logical strength [76, 77]. At the theory level, an interpolant is derived from the theory proof: for linear real arithmetic, the standard construction builds a Farkas interpolant from the coefficients of an infeasibility proof, while its different [6, 36] variants yield whole families of stronger or weaker interpolants, and the construction extends to tree interpolation [8].

Example 2. *Consider a following unsatisfiable logical formula:*

$$\overbrace{x < -10 \vee x = -1}^{A} \wedge \overbrace{x > 0}^{B}$$

For example, an interpolant I for this formula can be: $x < 0$.

In terms of complexity, computing an interpolant is inexpensive relative to solving: once a refutation proof is available, both the propositional and the theory-level constructions run in time linear in the size of that proof and produce an interpolant of comparable size. The interpolation cost lies in proof production, and the interpolant can, in the worst case, be exponentially larger than $A \wedge B$. Some theories also do not admit quantifier-free interpolants unless their signature is extended with auxiliary symbols, as is the case for the theory of arrays.

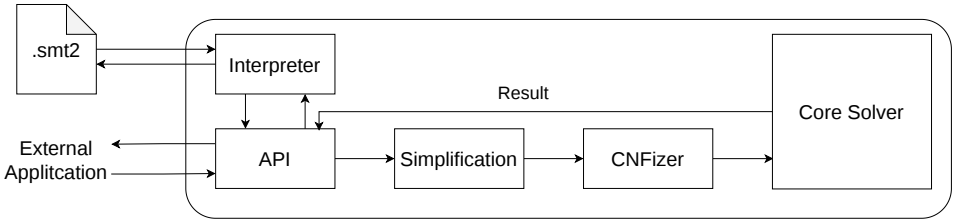


Figure 2.1. High-level architecture of OpenSMT.

2.1.3 OpenSMT Solver

There exist many different SMT solvers, which allow for quick and efficient SMT solving. The SMT solver we used in this thesis and to which development this thesis contributed is OpenSMT [53]. OpenSMT is an open-source SMT solver developed in the Formal Verification and Security Lab at the University of Lugano. It is available on GitHub[2]. The high-level architecture of the OpenSMT solver can be seen in Figure 2.1.

OpenSMT accepts two types of input: either an ".smt2" file or direct interaction with the solver through its API. The input instance is first simplified by removing tautologies and is then converted into CNF. Afterwards, it is handled by an internal solver composed of a SAT solver and multiple theory solvers. OpenSMT returns SAT or UNSAT, determining whether the given SMT instance is satisfiable, and it can additionally produce artefacts such as an interpolant, a variable assignment, or a proof of unsatisfiability.

OpenSMT supports various quantifier-free theories and theory combinations (LIA, UF, LRA, AX, and their combinations). It has consistently been evaluated as one of the best SMT solvers for the quantifier-free theories of LIA and LRA over many years according to the results of the SMT competition (including SMT-Comp 2026, 2025, 2024), which is a good match for the goal of software verification. OpenSMT natively supports interpolation over LIA, LRA, UF, and implements the Pudlak and McMillan interpolation algorithms, allowing to produce formulae of different strength. It is also capable of producing witnesses of satisfiability and unsatisfiability for logical formulae, which allows one to verify the SMT solving results.

[2] https://github.com/usi-verification-and-security/opensmt

2.2 Software Verification

2.2.1 Transition Systems

A transition system is a common formalism for modeling the behavior of software and hardware systems. In software verification, transition systems are used to represent programs. The states of a transition system correspond to program states, defined by the values of program variables at a specific moment in time and the location in the program. The transition encodes concrete execution steps of the system. This work particularly focuses on the symbolic representation of transition systems, where the states and transitions are represented by first-order logic formulae (using the SMT-lib format representation).

A transition system is defined over a set of fixed variables. X, X' is used to denote sets of variables (unprimed and primed), where X is a set of state variables, and $X' = \{x' | x \in X\}$ is a set of next state variables. If a formula contains variables over multiple states, the superscript notation is used, e.g., $X^{(0)}, \ldots, X^{(n)}$. The formula $F(X)$ defined over X is a state formula, and a formula $Tr(X, X')$ over $X \cup X'$ is a transition formula. State formulae encode a set of states, and transition formulae define binary relations over the states. $Id(X, X')$ is used to denote the identity transition formula $\bigwedge_{x \in X} x = x'$. A transition system can be defined as $\langle Init, Tr, X \rangle$, where $Init$ is a formula that encodes initial states, Tr is a transition formula, which represents the transition in the system, and X is a set of system's variables. A state s is a specific assignment of values to the variables in X. There exists a transition from the state s to the state s' if the formula $Tr(s, s')$ is satisfiable.

A state formula $F(X)$ holds for a state s if $s \models F$, i.e., $F(s)$ is true. The state s_n is called *reachable* within the transition system if there exists a sequence of states $s_0, s_1, \ldots, s_n$ such that $Init(s_0)$ (s_0 is an initial state), and for each $i \in [0, n-1]$ the formula $Tr(s_i, s_{i+1})$ is satisfiable. If such a sequence does not exist, the state s_n is called *unreachable*. Additionally, the concept of traces in a transition system is used in this thesis. *Trace* is a formula that describes multiple consecutive transitions $trace(X^{(0)}, X^{(1)}, \ldots, X^{(n)}) \triangleq Tr(X^{(0)}, X^{(1)}) \wedge \cdots \wedge Tr(X^{(n-1)}, X^{(n)})$.

One of the key concepts for transition systems is the notion of an invariant. A state formula $F(X)$ is an *invariant* if $F(s)$ holds for every reachable state s within the transition system. The invariant is called inductive if $\forall X, X'. F(X) \wedge Tr(X, X') \implies F(X')$. Intuitively, an invariant is a formula that overapproximates the reachable state space, and holds for every state reachable within the transition system. An inductive invariant additionally requires that the invariant is preserved by the transition relation.

2.2.2 Transition Invariants

One way to overapproximate the behavior of a transition system is via a transition invariant. A transition invariant is a formula that overapproximates any number of transitions within the transition system.

Definition 2 (Transition invariant)**.** *For a transition system $\langle Init, Tr, X\rangle$, $TrInv(X, X')$ is a transition invariant if: $\forall X, X'. R(X) \wedge Tr^{+}(X, X') \rightarrow TrInv(X, X')$, where $Tr^{+}(X, X')$ is the transitive closure of $Tr(X, X')$ and $R(X)$ is a formula encoding all states reachable in the transition system.*

A transition invariant can be produced by overapproximating the transition relation, and can be used to prove safety: if $Init(X) \wedge TrInv(X, X') \wedge Bad(X')$ is unsatisfiable, then no bad state is reachable from an initial state. The following property holds for transition invariants (where $R(X)$ is a formula representing all reachable states within the transition system):

$$\forall n \in \mathbb{N}. \forall X^{(0)}, X^{(1)}, \ldots, X^{(n)}. R(X^{(0)}) \wedge Tr(X^{(0)}, X^{(1)}) \wedge \cdots \wedge Tr(X^{(n-1)}, X^{(n)}) \rightarrow TrInv(X^{(0)}, X^{(n)})$$

2.2.3 Safety Verification

In the context of transition systems, the safety problem can be defined as $\langle Init, Tr, Bad, X\rangle$, where Bad is a formula that represents the states violating the safety property, and X is the set of system variables.

Example 3. *Example 3 demonstrates the TS-based encoding of the C program. As can be seen, the program is defined over variables x, y, l_0, l_1, e, where l_0, l_1, e are boolean variables introduced to denote a specific location in the program: l_0 corresponds to the first loophead, l_1 to the second, and e to the assert statement. Complex software systems are encoded using boolean or integer variables to denote specific program locations. The initial variable assignment is encoded as the initial states of the transition system: $Init \triangleq x = 0 \wedge y = 50 \wedge l_0$. The main program loops are encoded as the transition relation Tr. Finally, the assertion in the program represents the safety property that needs to hold during the execution of the program; the violation of the safety property is encoded as the negation of the condition inside the assertion: $Bad \triangleq e \wedge x \geq 200$.*

The goal of safety verification is to prove that there does not exist a reachable state s within the transition system that satisfies Bad. If there exists a satisfying trace from the initial states to a state that violates the safety property, then the transition system is unsafe:

$$\exists s_0, s_1, \ldots, s_n. Init(s_0) \wedge trace(s_0, s_1, \ldots, s_n) \wedge Bad(s_n)$$

C program

```
1  int x = 0;
2  int y = 50;
3  // $l_0$
4  while (x < 100) {
5    x = x + y;
6  }
7  // $l_1$
8  while (y > 0) {
9    y = y - x/10;
10 }
11 // $e$
12 assert(x < 200);
```

Safety Problem

$$Init \triangleq x = 0 \wedge y = 50 \wedge l_0$$

$$\begin{aligned} Tr \triangleq\ & (l_0 \wedge l_0' \wedge x < 100 \ \wedge\ x' = x + y \ \wedge\ y' = y) \vee \\ & (l_0 \wedge l_1' \wedge x \geq 100 \wedge x' = x \wedge y' = y) \vee \\ & (l_1 \wedge l_1' \wedge y > 0 \wedge x' = x \wedge y' = y - x/10) \vee \\ & (l_1 \wedge e' \wedge y \leq 0 \wedge x' = x \wedge y' = y) \end{aligned}$$

$$Bad \triangleq e \wedge x \geq 200$$

$$X \triangleq \{x, y\} \in \mathbb{Z}$$

Figure 2.2. C program and its symbolic encoding into a safety problem.

If no such trace exists, the transition system is safe. The absence of such a trace can be proven by the construction of an inductive invariant $Inv(X)$, such that:

$$Init(X) \rightarrow Inv(X)$$
$$Inv(X) \wedge Tr(X, X') \rightarrow Inv(X')$$

To prove safety, the invariant should also satisfy the following property: $\neg\exists X.Inv(X) \wedge Bad(X)$, meaning that $Inv(X) \wedge Bad(X)$ is unsatisfiable, and there does not exist a reachable state within the transition system that violates the safety property.

Safety verification of transition systems is a fundamental problem in formal methods, and has been extensively studied in the literature. Various techniques have been applied to TS verification, such as BMC, PDR, LAWI, TPA, and many others. The safety problem is closely related to the problem of invariant synthesis, as finding an inductive invariant proves the safety of the system.

2.2.4 Termination and Nontermination

While safety verification asks whether a bad state is reachable, termination analysis asks whether every execution of the system is finite. A *run* of a transition system $\langle Init, Tr, X \rangle$ is a finite or infinite sequence of states $s_0, s_1, \ldots$ such that $Init(s_0)$ holds and $Tr(s_i, s_{i+1})$ holds for every pair of consecutive states.

Definition 3 (Termination and nontermination)**.** *A transition system $\langle Init, Tr, X \rangle$ is* terminating *if it has no infinite run, i.e., there exists no infinite sequence of states $s_0, s_1, s_2, \ldots$*

such that

$$Init(s_0) \wedge \forall i \in \mathbb{N}.\ Tr(s_i, s_{i+1}).$$

Otherwise, i.e., if such an infinite sequence exists, the transition system is nonterminating.

For example, consider $X = \{pc, x\}$, $Init(X) \triangleq pc = 0$, and

$$Tr(X, X') \triangleq (pc = 0 \wedge pc' = 1 \wedge x' \geq 0) \vee (pc = 1 \wedge x > 0 \wedge pc' = 1 \wedge x' = x - 1).$$

The first transition nondeterministically chooses an arbitrary non-negative value of x, which is then decremented until it reaches 0. Every run of this system is finite, so the system is terminating, although from every initial state there exist runs of any length. Both termination and nontermination are undecidable in general, since they subsume the halting problem. Decidability results exist only for restricted classes, such as linear and some polynomial loops [45, 94, 108, 170].

Nontermination is typically witnessed by a *recurrent set* [105]: a set of states that contains an initial state and that execution never has to leave.

Definition 4 (Recurrent set)**.** *For a transition system $\langle Init, Tr, X\rangle$, a formula $N(X)$ is a recurrent set if: (1) $\forall X.N(X) \rightarrow \exists X'.Tr(X, X') \wedge N(X')$, and (2) $\exists X.N(X) \wedge Init(X)$.*

If a recurrent set exists, the transition system is nonterminating: starting from an initial state in N, an infinite run can be constructed by repeatedly choosing a successor that again lies in N.

Termination is witnessed by well-founded relations.

Definition 5 (Well-founded relation)**.** *A binary relation $W \subseteq D \times D$ over a domain D is* well-founded *if there exists no infinite descending chain:*

$$\nexists\, d_0, d_1, d_2, \cdots \in D.\ \forall i \in \mathbb{N}.\ (d_i, d_{i+1}) \in W$$

Definition 6 (Ranking function)**.** *Let $(D, \succ)$ be a well-founded domain, i.e., $\succ$ is a well-founded relation over D. A function f that maps states to D is a* ranking function *for a relation $W(X, X')$ if $\forall X, X'.\ W(X, X') \rightarrow f(X) \succ f(X')$.*

If a relation admits a ranking function, it is well-founded, since an infinite chain in W would induce an infinite descending chain in D. In practice, ranking functions are often linear functions $f(X) = \sum_i a_i x_i + c$ into the integers. Since $>$ is not well-founded over all integers, f must additionally be bounded from below on W:

$$\forall X, X'.\ W(X, X') \rightarrow f(X) \geq 0 \wedge f(X) > f(X').$$

Over the rationals, f must instead be bounded from below and decrease by at least a fixed $\delta > 0$ in every step [156]. A transition system terminates if its transition relation, restricted to the reachable states, admits a ranking function.

For complex systems, a single ranking function may not exist or may be hard to find. Podelski and Rybalchenko [157] showed that it suffices to cover the transitive closure of the transition relation by finitely many well-founded relations.

Definition 7 (Disjunctively well-founded relation)**.** *A relation $K(X,X') = \bigcup_{i=0}^{n} W_i(X,X')$ is called disjunctively well-founded if every relation $W_i(X,X')$ is well-founded.*

Theorem 1 ([157])**.** *A transition system $\langle Init, Tr, X\rangle$ is terminating if and only if there exists a transition invariant $TrInv(X,X')$ that is disjunctively well-founded.*

Note that the union of well-founded relations is not necessarily well-founded. The theorem relies on the fact that *TrInv* overapproximates the *transitive closure* of *Tr* (see Section 2.2.2), and not only *Tr* itself. The theorem is stated for transition invariants as relations over states.

Termination and nontermination analysis. The most widely studied approach to proving termination is the synthesis of ranking functions: complete methods exist for linear ranking functions of simple loops [22, 44, 156], and multiphase and lexicographic ranking functions broaden the class of programs that can be handled [21, 23]. Other approaches synthesize ranking functions via learning, syntax guidance, or decomposition [85, 109, 128, 163, 174]. Transition-invariant-based techniques construct disjunctively well-founded transition invariants, e.g., via abstraction refinement or loop summarization [65, 66, 127, 171], and can be exported as termination witnesses [27]. Fixpoint-based methods reduce both problems to the validity of first-order fixpoint formulae [173]. Nontermination analysis is centered on recurrent sets, constructed by constraint solving, Max-SMT, or symbolic execution [52, 105, 132]; alternatives include geometric nontermination arguments [134], loop acceleration [89, 90], and Hoare-style specification inference [133]. Several approaches reduce termination [30, 65] and nontermination [59] to safety verification, which allows them to reuse safety verifiers. This is also the starting point of the approach in Chapter 4. State-of-the-art termination tools include AProVE [96], T2 [51], and KoAT and LoAT [90, 137], which are compared annually in the Termination Competition [97, 98].

2.2.5 Constrained Horn Clause Solving

Constrained Horn Clauses (CHCs) are a powerful formalism for encoding safety problems. They allow for a more concrete and detailed representation of the original pro-

gram compared to transition systems, as constrained Horn clauses maintain the software structure.

A *constrained Horn clause* over a set of uninterpreted predicate symbols $R = \{r_1, \ldots, r_n\}$ is a universally quantified first-order formula of the form:

$$\forall X, X'.\ \bigwedge_{i=k}^{m} r_i(X) \wedge \varphi(X, X') \rightarrow h$$

where $r_k, \ldots, r_m \in R$, $\varphi(X, X')$ is an interpreted first-order logic formula, and h is either a single uninterpreted predicate $h(X) \in R$, or $\bot$. The left-hand side of the implication (called *body*) may have multiple occurrences of predicates from R, but φ contains no symbols from R. The right-hand side of the implication (called *head*) may contain at most one occurrence of a predicate from R. For readability, the leading $\forall X, X'.$ is omitted and all CHCs are assumed to be implicitly universally quantified.

This thesis works with a subset of CHCs — *Linear CHCs* — which allow at most one uninterpreted predicate in the body. A clause whose body contains no uninterpreted predicates is called a *fact*, and a clause with $\bot$ as the head is called a *query*.

Example 4. *In particular, the program in the Example 3 can be encoded as the following CHC system:*

$$\begin{aligned}
x = 0 \wedge y = 50 &\rightarrow L_0(x, y) \\
L_0(x, y) \wedge x < 100 \ \wedge\ x' = x + y \ \wedge\ y' = y &\rightarrow L_0(x', y') \\
L_0(x, y) \wedge x \geq 100 \wedge x' = x \wedge y' = y &\rightarrow L_1(x', y') \\
L_1(x, y) \wedge y > 0 \wedge x' = x \wedge y' = y - x/10 &\rightarrow L_1(x', y') \\
L_1(x, y) \wedge y \leq 0 \wedge x' = x \wedge y' = y &\rightarrow E(x', y') \\
E(x, y) \wedge x \geq 200 &\rightarrow \bot
\end{aligned}$$

As can be seen, the structure of the program is preserved in the CHC system. Predicates L_0 and L_1 correspond to the first and second loops of the C program. Each clause encodes the operations executed between loopheads. This structure allows for more precise reasoning that takes the program structure into account.

CHCs are used as an intermediate representation for procedural and object-oriented programs and allow the verification tools to symbolically enumerate program behaviors. CHC representation is beneficial for the purposes of software verification, as it decouples software modeling and software verification. As long as software can be

translated into a CHC representation, it can be submitted to any CHC solver to be verified.

The goal of CHC solving is similar to safety verification. CHC solvers seek an interpretation of all uninterpreted predicates in the system that makes every clause valid. This interpretation is analogous to constructing safe inductive invariants for program locations. CHC solvers return SAT if there exists such an interpretation, and UNSAT if it does not exist. The violation of the safety property is detected similarly to transition-system-based safety verification, by detecting a counterexample that allows to violate a safety property, called *unrolling*. An *unrolling* of a CHC system is a formula derived by repeatedly substituting clause bodies for predicate occurrences until no uninterpreted predicates remain. This concept corresponds to the concept of the trace in the transition system. If there exists an unrolling of the CHC system that yields a false statement, then the CHC solver returns UNSAT (meaning the safety property can be violated).

2.2.6 Quantifier Elimination and Model-Based Projection

Quantifiers are present in software verification for describing and reasoning about different properties, such as the existence of safe inductive invariants and transition invariants. However, reasoning about quantified formulae is significantly harder than in the quantifier-free case; one way to handle such formulae is to remove quantifiers from the formula. Quantifier elimination (QE) is the problem of deriving a quantifier-free formula equivalent to a given quantified formula within a given theory.

Example 5. *Consider the following formula:*

$$\exists y.\, -8 < y \wedge x + y < 0$$

Eliminating the existential quantifier: $\exists y.\, -8 < y \wedge y < -x$ *requires* $-7 < -x$*, i.e.,* $x < 7$*. The equivalent quantifier-free (in* y*) formula is therefore:* $x < 7$*.*

Quantifier elimination is computationally expensive, and many techniques use an underapproximation that computes only a partial result of QE. This underapproximation is produced using *Model-Based Projection* (MBP). Given an existentially quantified formula $\exists y.\, \varphi(x, y)$ and a model m satisfying φ, MBP returns a quantifier-free formula $\psi(x)$ such that $m \models \psi(x)$ and $\psi(x) \implies \exists y.\, \varphi(x, y)$. In other words, MBP produces a sufficient but not necessarily complete witness for the existential formula: it covers the model m but may not cover all satisfying assignments.

2.2.7 Golem CHC Solver

A large part of our work is built within the Golem Constrained Horn Clause Solver. Golem is a modular CHC solver implemented in the University of Lugano, which sup-

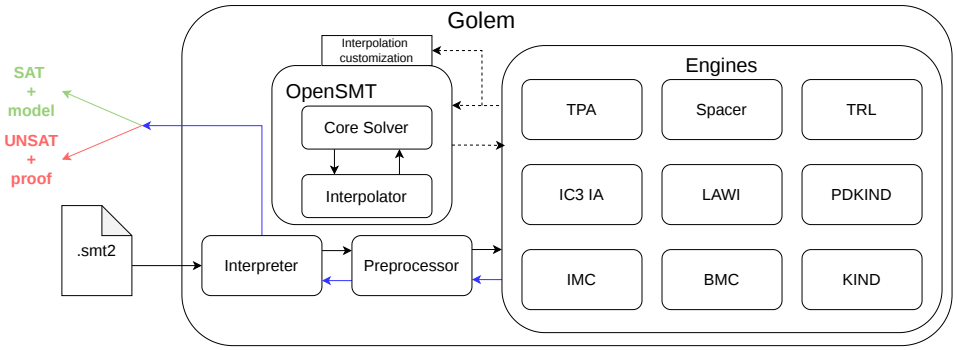


Figure 2.3. High-level architecture of Golem.

ports a range of techniques for CHC solving and model checking. It is available on GitHub[3]. It has consistently ranked among the top CHC solvers in the LIA, LRA, and LIA-Lin categories of CHC-Comp [71] (2023–2025), a competition for CHC solvers.

A key benefit of Golem is its tight integration with the underlying SMT solver, OpenSMT, which allows one to evaluate how changes in the underlying SMT solving engine influence verification. The modularity of the tool allows new CHC solving and model checking engines to be implemented and integrated efficiently. Additionally, Golem supports (non)termination analysis, has an implementation of QE and MBP, and support for witness generation, enabling validation of the verification results. The overall high-level architecture of Golem is depicted in Figure 2.3.

As can be seen in the figure, Golem receives a CHC system encoded in SMT-LIB2 format as input. This system is parsed into a graph representation and preprocessed, producing an equivalent simplified system. It is then solved by one or more CHC solving engines, using OpenSMT as the backend. Currently, Golem supports multiple CHC solving engines, including PDR, LAWI, and TPA. Golem returns SAT together with an interpretation of the uninterpreted predicates if such an interpretation exists. Otherwise, it returns UNSAT together with a counterexample unrolling witnessing the violation of the safety property.

2.3 Smart Contracts and Their Verification

Smart contracts are programs that are stored and executed on a blockchain. On Ethereum, the most widely used platform for smart contracts, contracts are compiled to bytecode of the Ethereum Virtual Machine (EVM), and the most popular source language is Solidity. A Solidity contract resembles a class in an object-oriented language: it consists

[3] https://github.com/usi-verification-and-security/golem

of state variables, which are kept in the persistent storage of the contract, and functions that operate on them. Users and other contracts interact with a deployed contract by sending *transactions* that call its public functions. Each transaction is executed atomically: it either completes and commits its changes to the contract state, or it *reverts*, undoing all of its effects. Besides the contract storage, a function can access the execution environment, such as the caller's address (`msg.sender`), the amount of Ether sent with the call (`msg.value`), account balances, and block data; every executed instruction consumes *gas*, which is paid for by the sender of the transaction.

Faults in smart contracts are particularly costly. Contracts often manage significant financial assets, their code is publicly visible, and once deployed it cannot be patched in place. As a result, vulnerabilities, such as reentrancy, arithmetic overflows, or unchecked external calls, have been repeatedly exploited, causing substantial losses [9]. This makes the analysis of contracts *before* deployment especially important.

From the verification perspective, a smart contract can be viewed as a transition system. Its state consists of the contract storage and the relevant part of the blockchain state, the initial states are determined by the constructor, and each transition corresponds to a call of one of the public functions, with arbitrary arguments and environment, in any order and any number of times. Properties of a contract, such as assertions in function bodies or invariants over the contract storage, are therefore safety properties over an unbounded number of transactions, while the function bodies themselves may contain loops and internal calls. This structure maps naturally to CHCs: one predicate captures the contract state between transactions (a contract invariant), and further predicates encode the summaries of individual functions [4, 139]. The encoding used in this thesis is described in Chapter 5.

A broad range of techniques has been applied to smart contracts. Static analyzers, such as Slither [86], detect known vulnerability patterns quickly, but are neither sound nor complete. Symbolic execution [11, 123], used for example by Oyente [138], explores contract paths with an SMT solver, but is limited by path explosion. Fuzzers, such as Echidna [104] and the fuzzer integrated into Foundry, execute contracts on large numbers of generated inputs [99]; they are efficient and produce concrete executions, but provide no guarantees of coverage. Formal verifiers prove properties of contracts: solc-verify [107] translates annotated contracts into a deductive verification language, VerX [155] verifies temporal safety properties, SmartACE [177] reuses LLVM-based analyzers on a translation of the contract, and SolCMC [4], the model checker built into the Solidity compiler, encodes contracts as CHCs and solves them with off-the-shelf CHC solvers. Formal verifiers return proofs or counterexamples, but not tests that developers can run and maintain. Chapter 5 builds on the CHC encoding of SolCMC to generate such tests automatically.

Chapter 3

Structure-aware Modular Safety Verification of Software

Model checking is one of the most active research fields within formal verification. Recent advancements in SMT, inductive invariant generation, and the introduction of new, efficient model checking techniques have significantly increased safety verification capabilities [24, 25]. Nonetheless, there is still a wide range of problems that require attention, such as model checking for nonlinear arithmetic, search for deep counterexamples, or analysis of multiple-loop systems.

A significant amount of research in model checking is centered around loop analysis. There exist many different approaches which target [88, 181] specifically single-loop programs. Techniques targeting multiloop and nested-loop approaches are less common, primarily because such systems are more difficult to analyze than single-loop software due to the complex inner structure, interconnected loops and branching. Nonetheless, developing such approaches is crucial, as such software is widespread.

One of the critical problems for multiloop analysis is the presence of deep loops which can have large number of iterations. The presence of such loops might significantly slow down model checking of the whole program. Recent work on TPA improves the analysis of deep loops. TPA was developed for reasoning about single-loop systems, however, it would be possible to apply it to multiloop programs using a transformation to merge multiple loops into a single loop [55]. Main drawback of this approach is that it loses structural information about the original program, leading to a slowdown of verification.

This chapter discusses a novel algorithm that enables effective reasoning over multiloop programs by applying TPA *modularly* and *incrementally* for each loop. It explores every possible execution path of the program, discovering *transition invariants* for each loop along a path and utilizing them during the exploration of other paths. Learned in-

formation about the safe states is propagated back and forth through the explored path, thus contributing to substantial runtime savings. This technique efficiently conducts safety verification for loops with large numbers of iterations leveraging the benefits of TPA. The algorithm handles programs with branching and multiple loops efficiently, as confirmed by experiments.

This chapter also presents an extension of the multiloop technique for verification of nested-loop software. This extension enables efficient verification of nested-loop systems while maintaining a modular and incremental verification strategy. It optimizes the multiloop technique using transition invariants as *loop summaries*. TPA is used to construct and refine loop summaries for every loop in the program, which enable reasoning about the safety of the overall system. The construction of the loop summaries is hierarchal and structural, as inner loop summaries are used to construct an outer loop summary, allowing for a more guided overapproximation. The experimental evaluation over nested-loop systems demonstrated the efficiency of the technique compared to the original multiloop approach as well as compared to other state-of-the-art techniques.

The two approaches were implemented inside the GOLEM CHC solver. They were experimentally compared to classical TPA (multiloop programs were transformed into single-loop in advance) and state-of-the-art tools, such as Z3-SPACER, ELDARICA, and LOAT [92]. Results demonstrate that the modular analysis is able to solve a significant number of multiloop and nested-loop benchmarks previously unmanageable both by GOLEM's competitors and by TPA. The original multiloop approach was published and presented at FM 2024 [47] and received the distinguished paper award. The extension for verification of nested-loop programs was published and presented at iFM 2025 [48]. Readers are referred to the Chapters 1 and 2 for notable references for some of the concepts, tools and algorithms used in this chapter.

3.1 Preliminaries

The approach is based on a symbolic program representation that maps the program's control flow to formulae in first-order logic. A set of logic formulae *Fla* is restricted to Linear Integer Arithmetic (LIA).

***While* language.** The presentation is restricted to programs in the conventional **While** language [150]. This language has the following sorts and categories: n (over integers), x (over variables), a (over arithmetic expressions), b (over boolean expressions),

```
1 int x1 = nondet(), x2 = nondet(), x3 = 0;
2 assume(0 <= x1 <= 100 && 0 <= x2 <= 50);
3 while (x1 < 300) { ++x1; ++x2; --x3;}
4 while (x3 < x2) { ++x3; --x1; }
5 assert(x1 <= 0);
```

Figure 3.1. Multiloop example.

and S (over statements):

$$
\begin{aligned}
a &::= n \mid \texttt{nondet()} \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2 \\
b &::= true \mid false \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg b \mid b_1 \wedge b_2 \\
S &::= \texttt{assert}(b) \mid \texttt{assume}(b) \mid x := a \mid \texttt{skip} \mid \\
&\quad S;S \mid \texttt{if } b \texttt{ then } S \texttt{ else } S \mid \texttt{while } b \texttt{ do } S
\end{aligned}
$$

Figure 3.1 gives an example of a program with two consecutive loops. It is presented in a more familiar **C** syntax, but it could easily be translated to **While**. Additionally, this example is used to illustrate the solving process of the algorithm in Section 3.2.3. The first loop increments both x_1 and x_2 until $x_1 \geq 300$, also decrementing x_3. The second loop increments x_3 and decrements x_1 until $x_2 \leq x_3$. The safety property of the program is given in the assertion $x_1 \leq 0$. This program is safe, as for any value of x_1 and x_2 the assertion will be satisfied. The example can be made unsafe by changing the assertion $x_1 \leq 0$ to $x_1 > 0$.

Program encoding and cutpoint graphs. The representation of the program assumes a global set of variables denoted X. Conventional primed notation is used to represent "next-state" variables. Multiloop programs are modeled using Cutpoint Graphs (CPG), which offer more compact program representations than classic Control Flow Graphs (CFG). Every node in a CPG (except *entry* and *error*) represents a loophead in the corresponding program. For every single loop-free segment between the loopheads, there exists a single corresponding edge in the CPG, even when there are multiple possible paths through it.

Definition 8. *Given a program $\mathscr{P}$, its* cutpoint graph $G_{\mathscr{P}} = \langle N, E, L, entry, error \rangle$ *is a tuple where N is a finite set of* cutpoints *(graph nodes), representing loopheads in the program. E is a set of actions between the cutpoints (edges between the nodes) of the form (u, v), where $u, v \in N \cup \{entry, error\}$. L is a mapping $L : E \to Fla$ from edges to logic formulae over X and X', representing symbolic encodings of loop-free statements, and entry and error are such that $\forall u \in N : (error, u) \notin E \wedge (u, entry) \notin E$.*

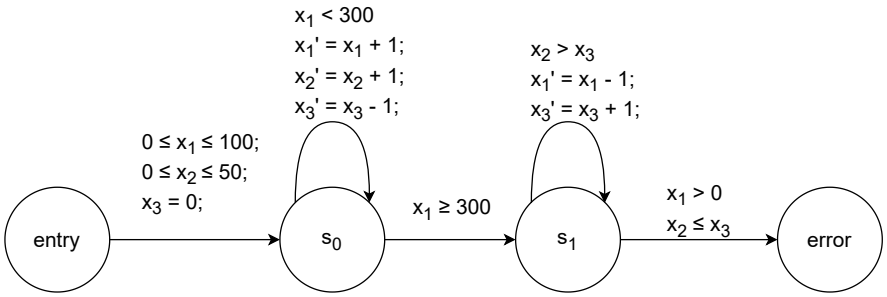


Figure 3.2. Cutpoint Graph for program in Figure 3.1. Transitions are labeled with constraints from Fla.

Based on Definition 8, it is possible to represent any program specified in **While** as a CPG. For example, the CPG for the program in Figure 3.1 is given in Figure 3.2. For a CPG $G_{\mathscr{P}} = \langle N, E, L, entry, error \rangle$, a *path* through the CPG is a sequence of edges $\pi = [e_1, e_2, \ldots, e_n]$, where $e_1, e_2, \ldots, e_n \in E$. **Path encoding** is a logical formula that encodes a path π within a cutpoint graph: $\phi(\pi) = L(e_1)(X^{(0)}, X^{(1)}) \wedge \ldots \wedge L(e_n)(X^{(n-1)}, X^{(n)})$. A path π is *feasible* if its path encoding $\phi(\pi)$ is satisfiable. Cutpoint graphs can be easily produced and translated from the linear Constrained Horn Clauses, converting the uninterpreted predicates to nodes in the CPG and the interpreted parts of the clauses into the edges labels. The clauses themselves correspond to the edges of the CPG - transition from one uninterpreted predicate to another.

Transition Power Abstraction. As mentioned in Chapter 1, TPA is an interpolation-based safety verification approach based on the abstraction of the transition relation. It takes a safety problem $\langle Init, Tr, Bad, X \rangle$ as input and decides if any Bad state is reachable from some initial state. Moreover, it can return a reflexive transition invariant if the system is safe or produce a provably reachable subset of Bad and a *trace* to reach it if the system is unsafe. A transition formula $TrInv(X, X')$ is a *reflexive transition invariant*, if $\forall X, X' : R(X) \wedge Tr^*(X, X') \implies TrInv(X, X')$, where Tr^* is the reflexive transitive closure of Tr and R is a set of reachable states in the system (meaning $TrInv$ is a transition invariant and $TrInv(X, X') \wedge \bigwedge_{x \in X} x = x'$ is satisfiable). The transition invariant $TrInv$ is *safe* if $Init(X) \wedge TrInv(X, X') \wedge Bad(X')$ is unsatisfiable.

One of the most important properties of the TPA is the ability to efficiently execute deep safety checks during the search for counterexamples. TPA runs iteratively, using transition abstractions $ATr^{\leq n}$, instead of exact transitions. $ATr^{\leq n}$ over-approximates the sequence of 2^n transitions for n-th iteration of the TPA, allowing it to double the amount of the considered transitions every iteration of the algorithm.

Example continued. The example program in Figure 3.1 has two interesting prop-

erties. First, the depth of the loops, which can overall result in up to 950 iterations. If it was an unsafe example, TPA would be more efficient than the approaches based on the explicit state abstraction due to its ability to overapproximate the transition relation. Second, in the presence of multiple loops, TPA would verify this program only if loops in the program are merged into a single loop. The merged loop will be bigger in size and lose structural information about the program, which could cause a slowdown in verification.

On the other hand, TPA could be applied to the loops separately. However, that would require some intermediate assertion to define safety property for the first loop and initial conditions for the second loop. For example, the condition $x_1 \leq x_2 - x_3$ could serve as such intermediate assertion introduced between lines 3 and 4. Notably, the algorithm presented in the next section *automatically* infers similar helper information and applies TPA modularly.

3.2 Multiloop Analysis with TPA

The algorithm performs forward reachability checking over the program's cutpoint graph. It searches for a feasible path from *entry* to *error*, building the path gradually and backtracking when the current path cannot be extended further. Before backtracking from a blocked state, it generalizes the reason for the conflict and learns blocking lemmas (similar to IC3/PDR-style algorithms). These define states that are guaranteed to be safe (i.e., there is no feasible path to *error* from these states), so the algorithm will know to avoid them the next time it reaches the same CPG node. Important to note that this the only type of loops in the CPG supported by this algorithm is a single self-looping edge loops. The proposed algorithm cannot handle complex nested loop systems (unless they are modelled as a single self-looping edge). This restriction is addressed in Section 3.4 where the algorithm is extended to handle nested loops. This contribution has been presented in FM 2024 [47].

3.2.1 Overview

To utilize the strengths of TPA, the algorithm alternates between two phases: i) reasoning about traversing from one loop to another, and ii) reasoning about traversing a single loop.

The first phase checks the feasibility of a single (large-block) step in the traditional sense, and it can be reduced to a single SMT check. However, the second phase attempts to extend the current path by getting to the exit of the current loop in an arbitrary number of its iterations. This effectively means checking a safety problem for

a transition system where initial states are the currently reached states, transition relation encodes one iteration of the loop, and error states are the states at the loop exit not yet blocked by the algorithm. While any algorithm for answering reachability queries over transition systems could be applied here, TPA has two advantages over traditional, state-focused model-checking algorithms. Its deep exploration makes it less likely to exhaust computational resources within a single loop that requires many iterations to traverse and, secondly, TPA is able to re-use *transition* invariants learned in previous queries to speed up current query to the same node. Note that when the algorithm reaches the same node but with a new state, the initial states (and possibly the error states too) of the safety problem change, but the transition relation always stays the same. Thus, the transition invariants from previous queries are still valid, while state invariants would very likely be invalidated.

3.2.2 The Core Algorithm

The algorithm for the verification of multiloop software is illustrated in Algorithm 1. It takes as input a CPG of a program together with a safety property and decides if the error node is reachable (UNSAFE) or not (SAFE). For each node v in the graph, the algorithm keeps track of two versions of the node, α_v and ω_v, called the *pre-state* and the *post-state*, respectively. The pre-state captures when the algorithm has reached v from another node. In programs, this represents execution reaching the loop head for the first time. The post-state captures when the algorithm is about to exit node v and continue to another node. In programs, this represents the execution exiting the loop. Each node version keeps track of a set of states already proven to be safe, denoted as $\alpha_v.safe$ and $\omega_v.safe$. These sets of states are represented as symbolic formulae initialized as $\bot$ (no states are proved safe at the start).

The algorithm also maintains the current feasible path prefix in the variable $path$ as a stack of entries of the form $\langle \alpha_v | \omega_v, \varphi \rangle$ representing that the set of states φ has been reached at the entry or exit of node v. At the beginning, $path$ is initialized as leaving the entry node with no restriction on the states (Line 1). When *error* is added to $path$, the algorithm has discovered a feasible path from *entry* to *error* and the program is unsafe (Line 17). If the algorithm ever backtracks beyond the initial entry ($path$ becomes empty), there is no feasible path from entry to error, and the program is safe (Line 27). Assuming $path$ is not empty, the algorithm attempts to extend the current feasible path prefix. There are two distinct cases. If the last entry on the path is a post-state of some node v (ω_v, Line 4), the algorithm attempts to use v's outgoing edges (ignoring the self-loop edge) to traverse to the pre-state of a *different* node w. Otherwise, the last entry is a pre-state of some node v (α_v, Line 20) and the algorithm attempts to get to the post-state of v by traversing v's self-looping edge some arbitrary

Algorithm 1: Multiloop-TPA

Input : Cutpoint Graph $G = \langle N, E, L, entry, error \rangle$;
Output: SAFE/UNSAFE

1 $path \leftarrow \langle \omega_{entry}, \top \rangle$
2 **while** $path$ *is not empty* **do**
3 **switch** *path.peek*() **do**
4 **case** $\langle \omega_v, curr \rangle$ **do**
5 **if** $\forall w \in N,\ s.t.\ (v,w) \in E : (v,w).blocked \neq \bot$ **then**
6 $blocked \leftarrow \bigwedge_{e \in v.outgoing} e.blocked$
7 $\omega_v.safe \leftarrow \omega_v.safe \vee \bigwedge_{e \in v.outgoing} e.blocked$
8 $path.pop()$
9 **for** $e \in v.outgoing$ **do** $e.blocked \leftarrow \bot$
10 **continue**
11 **else**
12 pick $w \in N$ s.t. $(v,w) \in E$ and $(v,w).blocked = \bot$
13 $t(X,X') \leftarrow curr(X) \wedge L((v,w))(X,X') \wedge \neg(\alpha_w.safe)(X')$
14 **if** $SAT?[t(X,X')]$ **then** `// TraverseBridge`
15 let $M \models t(X,X')$
16 $path.push(\langle \alpha_w, MBP(\exists X : t(X,X'), M) \rangle)$
17 **if** $w = error$ **then return** UNSAFE
18 **else**
19 $(v,w).blocked \leftarrow Itp(curr(X), L((v,w))(X,X') \wedge \neg(\alpha_w.safe)(X'))$
20 **case** $\langle \alpha_v, curr \rangle$ **do** `// TraverseLoop`
21 $(res, reached, TrInv) = \text{TPA}(curr(X), L((v,v))(X,X'), \neg(\omega_v.safe)(X'))$
22 **if** $res = reachable$ **then**
23 $path.push(\langle \omega_v, reached \rangle)$
24 **else**
25 $\alpha_v.safe \leftarrow \alpha_v.safe \vee Itp(curr(X), TrInv(X,X') \wedge \neg(\omega_v.safe)(X'))$
26 $path.pop()$
27 **return** SAFE

number of times. These two cases are described in detail below.

Post: When the algorithm is leaving some node v with reached states *curr* (Line 4), it searches for an unblocked outgoing edge as a candidate for extending the current path prefix (Line 5). An edge e is marked as blocked if the current path prefix cannot be extended with this edge, and the algorithm remembers the set of blocked source states (states for which it is not feasible to traverse the edge) in $e.blocked$. Algorithm 1 ensures the blocked states are superset of *curr*.

If all outgoing edges are blocked, it means that all outgoing edges have been considered as possible extensions, but all have failed eventually. The current path thus cannot be extended, and the algorithm backtracks to the pre-state of v (Line 8) to try a different continuation from that point. Before backtracking, the algorithm learns a new set of safe states as the intersection of states that are safe for individual outgoing edges (these are guaranteed to include all currently reached states) and unblocks all edges (Lines 7–9).

If, on the other hand, there is an unblocked edge (Line 12), the algorithm attempts to reach some potentially unsafe state of the edge's target node. The feasibility of this traversal, given the constraint of the edge, is checked in Lines 14-19 which, for simplicity, are called `TraverseBridge` throughout the rest of the chapter. It decides if some potentially unsafe states are reachable and computes a set of definitely reached target states (in case of reachability) or a set of definitely blocked source states (in case of unreachability). If the traversal is feasible, the path is extended (Line 16), and the analysis will continue from the new reached point unless *error* has been reached, in which case the algorithm immediately terminates (Line 17). If the traversal is infeasible, the picked edge is blocked (Line 19), marked with superset of *curr* for which the traversal is infeasible (see more details on `TraverseBridge` below). In the next iteration, the algorithm tries to pick a different, unblocked edge.

Pre: When the algorithm is entering some node v with reached states *curr* (Line 20), it attempts to find a feasible traversal of the loop, i.e., to reach some potentially unsafe post-state of v (taking an arbitrary number of loop iterations). The feasibility of this traversal is checked in Lines 21-26, which for simplicity are called `TraverseLoop`. Similarly to `TraverseBridge`, `TraverseLoop` not only decides the feasibility of the traversal but also computes a set of definitely reached target states to extend the current path (Line 23) or a set of definitely blocked source states, which forces backtracking (Line 26). Further details on `TraverseBridge` and `TraverseLoop` are provided below.

`TraverseBridge`. Given reached states *curr*, target states $\neg(\alpha_w.safe)$, and a transition constraint $L((v,w))$, the goal is to check if any target states are reachable from source states with *one* step of the transition constraint. The reachability check then

amounts to the satisfiability check for the conjunction of the three formulae (denoted t to simplify writing). Provably reached state can be defined exactly as $\exists X : \varphi(X)$. To avoid quantifiers, the set of reached states is underapproximated with *model-based projection* (MBP). The formula: $\neg\exists X' : L((v,w))(X,X') \wedge \neg(\alpha_w.safe)(X')$ describes provably blocked states. It is possible to avoid quantifiers, obtaining a generalization of the source states, using Craig interpolation.

`TraverseLoop`. Given reached states *curr*, target states $\neg(\omega_v.safe)$, and a transition constraint $L((v,v))$, the goal is to check if any target states are reachable from source states with *any number of steps* of $L((v,v))$. This is equivalent to deciding a safety problem for a transition system $\langle Init, Tr, Bad\rangle$ with $Init = curr$, $Tr = L((v,v))$ and $Bad = \neg(\omega_v.safe)$. TPA satisfies the additional requirements on `TraverseLoop`. It internally computes provably reached states as part of the witness for reachability. Provably blocked states can be computed using a safe transition invariant that TPA computes as a witness for safety. Similarly to `TraverseBridge`, Craig interpolation is leveraged to eliminate quantifiers. Note that computing a logically weak (more general) interpolant for $A = curr$ and $B = TrInv \wedge \neg(\omega_v.safe)$ yields a potentially much larger set of blocked states than the source states themselves.

Using TPA for implementing `TraverseLoop` has the additional advantage that TPA learns bounded and unbounded *transition* invariants during a single reachability check, which can be leveraged to bootstrap the transition abstractions in future reachability queries for the same loop. The ability of not starting from scratch has the potential to significantly speed up the consequent queries.

3.2.3 Running Example

To demonstrate the execution of Algorithm 1, the motivating example from Figure 3.2 is used as input. The execution is then depicted in Figure 3.3.

Initially, the algorithm attempts to leave *entry* and picks the single (unblocked) outgoing edge leading to s_0. Potentially unsafe states at α_{s_0} are $\top$ at this point, so `TraverseBridge` computes reached states at α_{s_0} to be $0 \leq x_1 \leq 100 \wedge 0 \leq x_2 \leq 50 \wedge x_3 = 0$.

As the next step, the algorithm attempts to traverse loop s_0 with $\top$ as the potentially unsafe states at ω_{s_0}. `TraverseLoop` determines that with one loop iteration state $1 \leq x_1 \leq 101 \wedge 1 \leq x_2 \leq 51 \wedge x_3 = -1$ is reached at ω_{s_0}.

Attempting to continue from this state will now fail, because the only outgoing edge to s_1 is not feasible, as determined by `TraverseBridge` with $x_1 < 300$ being the blocked states.

The algorithm now backtracks to α_{s_0} and attempts to traverse loop s_0 again, but this time with $x_1 \geq 300$ as the potentially unsafe states. Here TPA quickly determines that those states are reachable, e.g. after 255 iterations of the loop the state $x_1 = 300 \wedge 255 \leq x_2 \leq 305 \wedge x_3 = -255$ is reached.

From this state at ω_{s_0} it is possible to traverse to α_{s_1}, reaching states defined by the same formula. Next, the algorithm attempts to traverse loop s_1.

Similarly to how it behaved for the first loop, TPA suggests exiting the second loop after one iteration, in state $x_1 = 299 \wedge 255 \leq x_2 \leq 305 \wedge x_3 = -254$. However, when checking the single outgoing edge to *error*, `TraverseBridge` determines the infeasibility of this attempt and computes $x_2 > x_3$ as safe states at ω_{s_1}.

Thus, the algorithm backtracks again and attempts to traverse loop s_1 in a different way so that it ends up in a potentially unsafe state $x_2 \leq x_3$. Here TPA quickly determines that such a state can be reached after 511 iterations, with variable values $x_1 = -211 \wedge x_2 = 256 \wedge x_3 = 256$. However, this path cannot reach *error*, as determined by `TraverseBridge` with $x_1 \leq 0$ determined to be safe states at ω_{s_1}.

In the final attempt to traverse s_1 TPA determines that no unsafe state is reachable anymore and computes $x_1 \leq 384 \wedge 384 \leq x_2 - x_3$ as new safe states at α_{s_1}. Thus, the algorithm backtracks again and tries to find a different way to reach unsafe states of α_{s_1} from ω_{s_0}. `TraverseBridge` determines this to be impossible with $x_1 \leq 384 \wedge 384 \leq x_2 - x_3$ being safe at ω_{s_0} as well. Note that this condition can be viewed as an intermediate assertion between the two loops (as briefly mentioned in Section 3.1). It is sufficient to prove *error* cannot be reached by traversing the second loop, and, as will be seen shortly, it cannot be violated by traversing the first loop.

Finally, after backtracking to α_{s_0}, an attempt to traverse loop s_0 to avoid the safe states at ω_{s_0} fails, as TPA in `TraverseLoop` determines that $x_1 \leq 108 \wedge x_3 - x_2 \leq 0$ are safe states at α_{s_0}. Finally, the algorithm backtracks to ω_{entry}, and, with no new feasible way to extend the path, it concludes safety.

3.2.4 Correctness

This section first proves correctness when Algorithm 1 answers UNSAFE.

Theorem 2. *When Algorithm 1 returns* UNSAFE*, there exists a feasible path from entry to error.*

Proof. By induction on the entries of $path$, for every entry $\langle v, \varphi \rangle$ added to $path$, states φ at node v are reachable from *entry*.

Base case. The claim trivially holds for the initial entry $\langle \omega_{entry}, \top \rangle$ added on Line 1, since *entry* is reachable from itself.

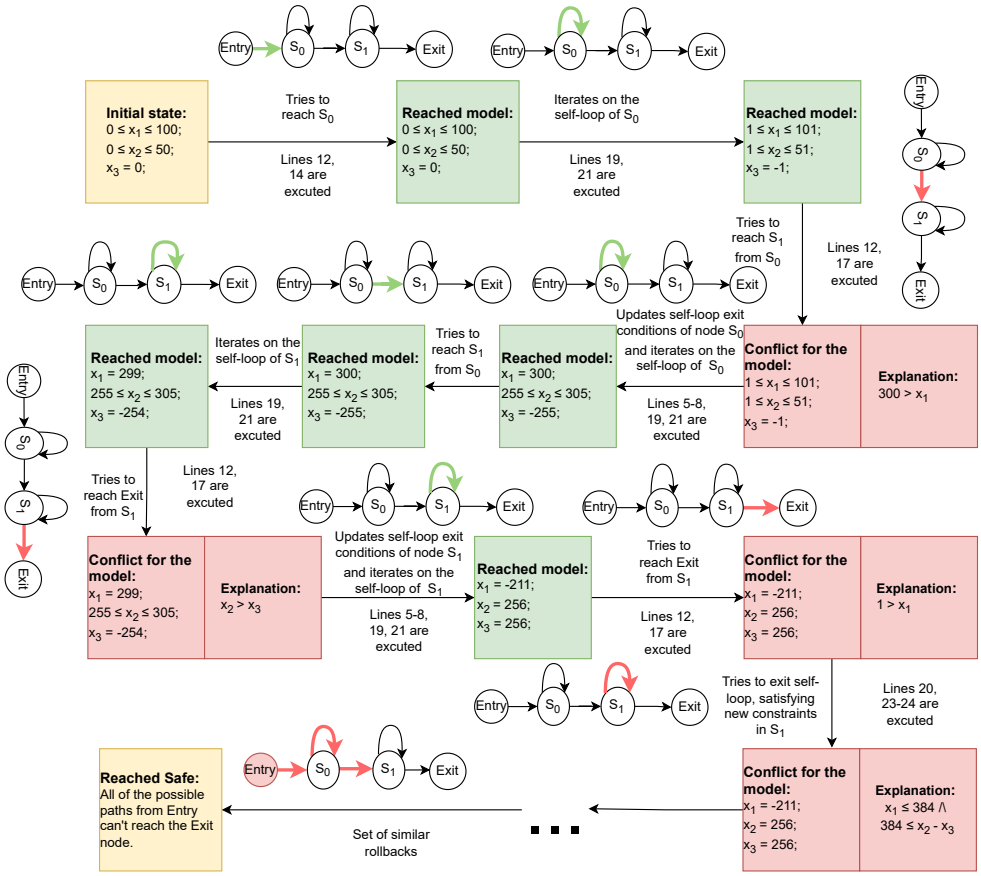


Figure 3.3. Algorithm execution flow for the example from Figure 3.2.

Inductive step. New entries are added to $path$ only at Lines 16 and 23, in both cases as an extension of the entry currently on top of $path$. By the induction hypothesis, the states of that entry are reachable from *entry*. It follows from the properties of `TraverseBridge` and `TraverseLoop` that the newly reached states pushed onto $path$ are reachable from the states of the previous top entry, and hence, by transitivity, from *entry*.

Algorithm 1 returns UNSAFE exactly when an entry $\langle error, \cdot \rangle$ is pushed onto $path$ (Line 17); by the above, the sequence of entries in $path$ at that point witnesses a feasible path from *entry* to *error*. □

The correctness of the safe answer is proved next, using some auxiliary lemmas.

Lemma 1. *The following is an invariant of the algorithm: For each node $v \in N$ and each state $s \in v.safe$ there is no feasible path from entry to error going through $\langle v, s \rangle$.*

Proof. Initially, all sets of safe states are empty ($\bot$), so the invariant holds trivially. Sets of safe states are extended at two points: Line 7 and Line 25.

On Line 25, the set of safe states for node α_v is extended with the blocked states from `TraverseLoop`. `TraverseLoop` ensures that the blocked states are a superset of the currently reached states in α_v that is guaranteed to only reach safe states of ω_v. Thus, this extension of safe states preserves the invariant.

On Line 7, the set of safe states for node ω_v is extended with the intersection of the blocked states computed for each of v's outgoing edges. `TraverseBridge` ensures that the blocked states computed for an edge are a superset of the currently reached states in ω_v that is guaranteed to only reach safe states of w, the target of the edge. Thus, this extension of safe states preserves the invariant, too. □

Lemma 2. *When an entry $\langle v, \varphi \rangle$ is about to be popped from $path$ (Lines 8 and 26), the current path* cannot *be extended to a feasible path from entry to error.*

Proof. The entry $\langle v, \varphi \rangle$ is popped on Line 8 (resp. Line 26) precisely when a superset of φ has just been added to $\omega_v.safe$ (resp. $\alpha_v.safe$) on Line 7 (resp. Line 25), so $\varphi \in \omega_v.safe$ (resp. $\varphi \in \alpha_v.safe$) at the moment of popping. By Lemma 1, there is no feasible path from *entry* to *error* through any state in $\omega_v.safe$ or $\alpha_v.safe$, and in particular not through $\langle v, \varphi \rangle$. Therefore, the current path prefix ending in $\langle v, \varphi \rangle$ cannot be extended to a feasible path from *entry* to *error*. □

Theorem 3. *When Algorithm 1 returns* SAFE*, there is no feasible path from entry to error.*

Proof. Algorithm 1 returns SAFE exactly when the initial entry $\langle entry, \top \rangle$ is popped from $path$ (Line 27). By Lemma 2, this means the current (empty) path prefix, corresponding to the unrestricted initial states $\top$, cannot be extended to a feasible path from *entry* to *error*. Hence, no feasible path from *entry* to *error* exists. □

3.2.5 Witness Production

This section shows that Algorithm 1 can be extended to produce witnesses for both safe and unsafe programs (if it terminates).

Violation witnesses. A witness can be computed from $path$ constructed by Algorithm 1, following the standard notion of a violation witness as a counterexample path defined by a sequence of program states.

Definition 9 (Violation witness). *Given a CPG $G_{\mathscr{S}} = \langle N, E, L, entry, error\rangle$, a violation witness is an execution path $[s_1, ..., s_n]$ such that*

- *for each $i \in [1, n]$, tuple $s_i = \langle v_i, st_i\rangle$ where $v_i \in N$ and st_i is a program state, i.e., an assignment of all program variables,*
- *$s_1 = \langle entry, \top\rangle$ and $s_n = \langle error, q\rangle$ for some $q \neq \bot$,*
- *for each consecutive pair $\langle v_i, st_i\rangle$ and $\langle v_{i+1}, st_{i+1}\rangle$, $(v_i, v_{i+1}) \in E$ and $L((v_i, v_{i+1}))(st_i, st_{i+1})$ is satisfiable.*

In a violation witness, loops are represented explicitly by their iterations: each iteration of a loop v corresponds to two consecutive states $\langle v, st_i\rangle$ and $\langle v, st_{i+1}\rangle$ of the witness, connected by the self-loop edge (v, v). A witness that traverses loop v in k iterations therefore contains $k + 1$ consecutive states at node v.

When Algorithm 1 decides the input CPG to be unsafe (Line 17), the entries in $path$ form a *blueprint* for the violation witness. Let $entry = v_0, v_1, \ldots, v_m = error$ be the sequence of nodes recorded in $path$. The blueprint defines exactly which loops the counterexample traverses and in what order. However, it does not define how many iterations are taken in each loop, nor the intermediate states of the program in those iterations.

This information is collected during the traversal of the CPG. Every time `TraverseLoop` succeeds for a node v_i (Line 23), TPA has found a counterexample for the reachability problem $\langle curr, L((v_i, v_i)), \neg\omega_{v_i}.safe\rangle$: a trace of some length k_i that leads from a state in *curr* to a state in $\neg\omega_{v_i}.safe$. The provably reached states *reached* correspond to this trace, i.e., every state in *reached* is reached from *curr* in exactly k_i iterations of the loop. The algorithm stores the length k_i together with the entry $\langle\omega_{v_i}, reached\rangle$ pushed to $path$. If the algorithm later backtracks over this entry, k_i is discarded with it, and if the loop is traversed again from different reached states, a new length is computed. Consequently, when *error* is reached, every loop v_i on the path in $path$ is annotated with the number of iterations k_i of its most recent successful traversal. Note that only these numbers are stored, not the traces themselves.

The blueprint together with the iteration counts defines a concrete path through the CPG: the edge (v_0, v_1), followed by k_1 repetitions of the self-loop edge (v_1, v_1), followed by the edge (v_1, v_2), and so on, until the edge (v_{m-1}, v_m).

Constructing the witness thus requires only a single SMT query over a loop-free formula after the verification has finished. The size of formula, grows linearly with the total number of loop iterations $\sum_i k_i$, which can be large, since TPA is designed to discover long counterexamples. This is inherent to violation witnesses that list every program state explicitly.

Safety witnesses. Inductive invariants are used as safety witnesses.

Definition 10 (Safety witness). *Given a CPG* $G_{\mathscr{S}} = \langle N, E, L, entry, error \rangle$*, a safety witness is a mapping* $Inv : N \mapsto Fla$ *from loops to state formulae such that* $Inv_{entry} = \top$*,* $Inv_{error} = \bot$*, and* $\forall (v,u) = e \in E : Inv_v(X) \wedge L(e)(X,X') \implies Inv_u(X')$.

Note that this definition includes the requirement that Inv_v is an inductive invariant because the condition must hold also for self-loop edges (v,v).

Inductive invariants can be computed from the information produced by Algorithm 1. Recall that the algorithm computes for each loop v the set of safe states $\alpha_v.safe$ and $\omega_v.safe$. A safety witness can be obtained by computing, separately for each loop v, a safe inductive invariant for a reachability problem $\langle \alpha_v.safe, L((v,v)), \neg\omega_v.safe \rangle$ (which is known to be safe).[1]

Lemma 3. *Suppose* Inv_v *is a safe inductive invariant for a reachability problem* $\langle \alpha_v.safe, L((v,v)), \neg\omega_v.safe \rangle$ *for all* $v \in N$*. Then invariants of the CPG nodes* Inv_v *is a safety witness according to Definition 10, i.e.,* $\forall (u,v) = e \in E : Inv_u(X) \wedge L(e)(X,X') \implies Inv_v(X')$.

Proof. Each $Inv_v(X)$ is, by construction, an inductive invariant for its corresponding loop v. It remains to show that these invariants are inductive also with respect to transitions between loops.

Consider an edge $e = (u,v)$ with $u \neq v$. Since Inv_u is a safe inductive invariant for the reachability problem $\langle \alpha_u.safe, L((u,u)), \neg\omega_u.safe \rangle$, it follows that $Inv_u \implies \omega_u.safe$. Moreover, $\omega_u.safe \wedge L(e) \implies \alpha_v.safe$ is valid based on how the set of safe states is constructed in Algorithm 1: only those states at u that cannot reach states outside of $\alpha_v.safe$ are ever added to $\omega_u.safe$. Finally, $\alpha_v.safe \implies Inv_v$ is valid by construction of $Inv_v(X')$ as the inductive invariant for $\langle \alpha_v.safe, L((v,v)), \neg\omega_v.safe \rangle$. All three implications together yield the desired property $Inv_u \wedge L(e) \implies Inv_v$.

□

3.3 Evaluation of the Multiloop Verification

Algorithm 1 has been implemented in the Golem CHC solver[2]. This implementation is referred to as Golem-Multiloop. In the experiments, Golem-Multiloop is compared with state-of-the-art tools Z3-Spacer (v4.13.0) and Eldarica (v2.1.0), as well as the existing TPA and Spacer engines of Golem (denoted as Golem-TPA and Golem-Spacer). Benchmarks are centered specifically around multiloop instances. All experiments were conducted on a machine with an AMD EPYC 7452 32-core processor

[1] Any model checking algorithm can be used here, including TPA.

[2] https://github.com/usi-verification-and-security/golem.

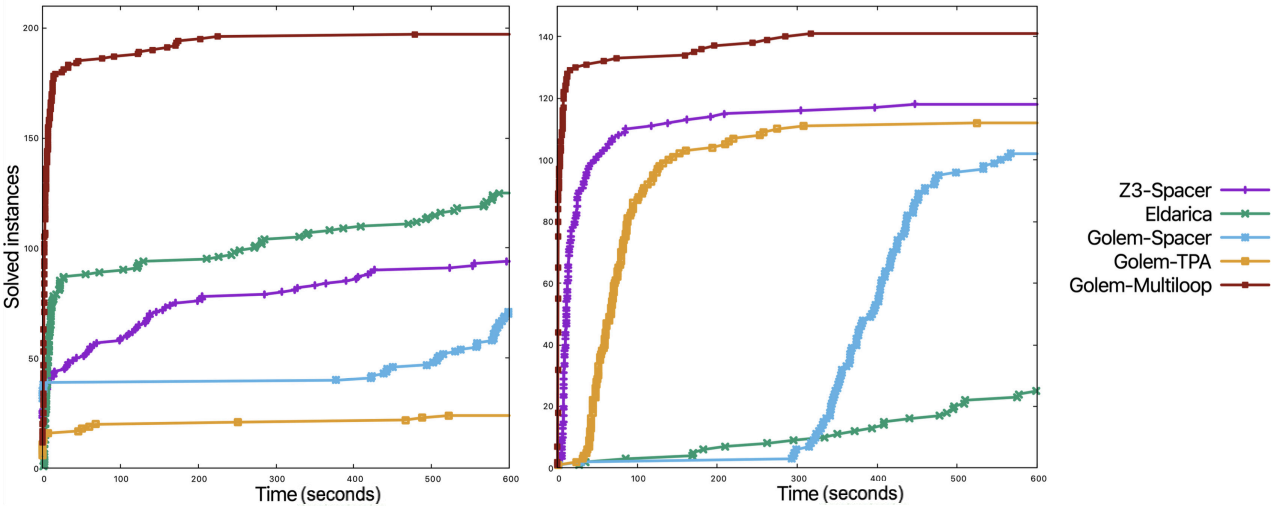


Figure 3.4. Comparison of performance of Golem-Multiloop with other tools: Z3-Spacer, Eldarica, Golem-Spacer and Golem-TPA. Plot on the left demonstrates the number of solved safe instances over time, plot on the right shows UNSAFE instances.

and 8x32 GiB of memory (the artefact, including the benchmarks and an executable version of GOLEM-MULTILOOP, is publicly available[3]).

The evaluation aims to answer the following two research questions:

- **RQ1:** How does the new *modular* algorithm compare to GOLEM-TPA running on a transformed single-loop program?
- **RQ2:** How does the performance of GOLEM-MULTILOOP fare against state-of-the-art tools?

The set of benchmarks[4] used in the experiments is partially composed of SV-COMP-23 instances[5] (specifically from the 'loops-crafted-1' set) and partially of crafted multiloop examples. The benchmarks have a common structure, with multiple loops interconnected between each other without nested loops. The motivating example from Figure 3.1 illustrates the structure of these benchmarks. The benchmark set consists of 263 safe and 179 unsafe problems.

Quantile plots, shown in Figure 3.4, compare the performance of individual tools on the benchmark set. A data point (x, y) in the plot represents the fact that the corresponding algorithm solved y problems given time x (in seconds). The results

[3] https://zenodo.org/records/18683197.
[4] https://github.com/BritikovKI/fv-benchmarks-2024
[5] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

show that GOLEM-MULTILOOP outperforms GOLEM-TPA both for safe and unsafe instances. The large performance improvement for safe instances can be attributed to GOLEM-MULTILOOP's modularity. While GOLEM-TPA has to find a single safe transition invariant for the whole (transformed) program, GOLEM-MULTILOOP builds separate transition invariants for individual loops *incrementally*.

For unsafe problems, the difference between the two approaches is smaller but still significant. The modular nature of GOLEM-MULTILOOP likely also helps it build better, more focused transition abstractions which, in turn, allow it to discover the real counterexample faster than GOLEM-TPA, which needs to spend more time refining the abstraction of the monolithic transition relation. To answer RQ1, the incremental and modular nature of GOLEM-MULTILOOP delivers significant improvements over applying TPA in a monolithic way to a transformed single-loop program.

GOLEM-MULTILOOP also significantly outperforms state-of-the-art tools. From the safe problems, GOLEM-MULTILOOP solves 198 benchmarks, while the second best, ELDARICA, was able to solve 125 benchmarks. However, ELDARICA solved 10 safe instances uniquely, demonstrating some orthogonality to the approach. Similar results can be observed for the unsafe benchmarks. GOLEM-MULTILOOP solves 23 instances more than Z3-SPACER, the second-best tool, even though Z3-SPACER was able to solve 12 instances uniquely. To answer RQ2, the evaluation shows that GOLEM-MULTILOOP significantly improves upon the state of the art, solving more instances than the next-best competitor. Moreover, it is on average 4.1 times and 2.8 times faster than the next best competitor on unsafe and safe instances, respectively.

Overall, the evaluation demonstrates that the new algorithm is capable of successfully handling both safe and unsafe challenging multiloop programs. It significantly improves not only over TPA applied to transformed single-loop programs but also over existing state-of-the-art tools.

3.4 The Multiloop Approach Extended to Nested Loops

The multiloop approach presented above is not efficient for verification of software with nested loops, as it cannot handle complex nested structure, and can only handle systems where nested loops are encoded as a single transition. The remainder of this section presents an approach that handles nested **While** programs, allowing complex loop structure. This technique uses a similar exploration flow to the multiloop approach presented in Section 3.4.3. It applies TPA to build and refine the loop summaries during verification, which are then used to overapproximate inner loop execution. This additional loop approximation-based strategy enables more effective reasoning about the behavior of nested loops. This contribution has been published and presented at

```
1 int x1 = 0, x2 = 0;
2 while (x1 < 512) {
3  while(x2 < 32) { x2 = x2 + 2; }
4  x1 = x1 + x2; x2 = 0;
5 }
6 assert(x1 == 512);
```

Figure 3.5. Nested loop example.

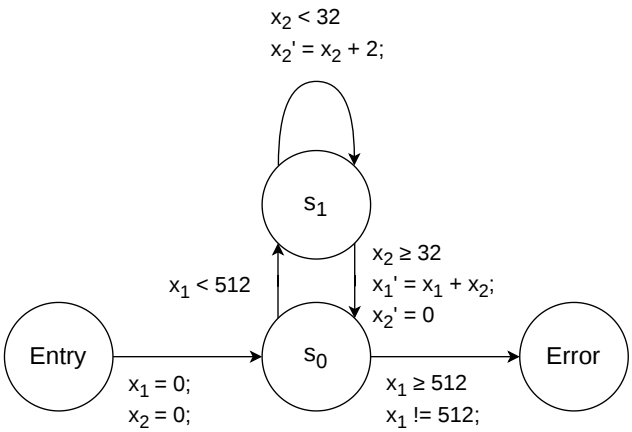


Figure 3.6. Cutpoint Graph for program in Figure 3.5. Transitions are labeled with constraints from *Fla*.

iFM 2025 [48].

3.4.1 Nested Loops and Loop Summaries

The approach uses a cutpoint graph to represent programs with nested loops, similarly to how multiloop programs are represented in Section 3.4.3. Consider the **While** language program in Figure 3.5, written in C syntax.

This program contains two loops: an outer loop at line 2 and an inner loop at line 3. The outer loop increments x1 by the value of x2 and assigns 0 to x2, while the inner loop increments x2 by 2 while x2 $<$ 32. When execution of the loops is finished, x1 equals 512, so the program is safe. The CPG representation of this program can be seen in Figure 3.6.

Inner loop. For nodes $a, b \in N$ of a CPG $G_{\mathscr{P}} = \langle N, E, L, entry, error \rangle$, a *path from* a *to* b is a sequence of edges $\pi = [(v_0, v_1), (v_1, v_2), \cdots, (v_{k-1}, v_k)]$ with $k \geq 1$, $v_0 = a$,

$v_k = b$, and $(v_{i-1}, v_i) \in E$ for every $i \in [1,k]$. The set of nodes it visits is denoted $nodes(\pi) = \{v_0, v_1, \ldots, v_k\}$.

Definition 11 (Inner loop node). *Let $G_{\mathscr{P}} = \langle N, E, L, entry, error\rangle$ be a CPG and let $v, u \in N$. A node $u \neq v$ is an* inner loop node *of v if u is reachable from entry, error is reachable from u, and v occurs on every connecting path:*

$$\forall\, \pi \text{ from entry to } u.\ v \in nodes(\pi) \qquad \text{and} \qquad \forall\, \pi \text{ from } u \text{ to error}.\ v \in nodes(\pi).$$

The set of all inner loop nodes of v is denoted $Inner(v) \subseteq N$.

In a corresponding manner, a node u is an *outer loop node* of v if v is an inner loop node of u.

Loop Summary For a CPG $G_{\mathscr{P}} = \langle N, E, L, entry, error\rangle$ Loop Summary is a formula $\Sigma_v(X, X')$, where $v \in N$ is a node in the cutpoint graph, representing a specific loop-head. $\Sigma_v(X, X')$ overapproximates the encoding of any possible path from node v to itself which contains edges through inner loop nodes or the node v itself. For example, for loop s_1 in Figure 3.5 the corresponding loop summary $\Sigma_{s_1}(X, X') = (x_2 < 32 \wedge x_2' \leq 33 \wedge x_1' = x_1) \vee (x_2 \geq 32)$. This formula overapproximates any possible number of iterations of loop s_1.

Paths through the CPG. Reasoning about the behavior of the nested loops requires the notions of iteration paths and simple paths through the CPG. These paths are used to construct the loop summaries.

Definition 12 (Iteration path). *Let $G_{\mathscr{P}} = \langle N, E, L, entry, error\rangle$ be a CPG and let $v \in N$. A path $\pi = (v_0, v_1)(v_1, v_2)\cdots(v_{k-1}, v_k)$ from v to v (so $v_0 = v_k$) is an* iteration path *of v if its first edge is its only edge with source v: $\forall\, i \in [1,k].\quad v_i = v_0 \implies i = k$*

Definition 13 (Simple path). *Path $\pi = [(v,u), \ldots, (k,l)], (v,u), \ldots, (k,l) \in E$ through the CPG $G_{\mathscr{P}} = \langle N, E, L, entry, error\rangle$ is called* simple *if $\forall (v,u) \in \pi. \neg\exists (k,l) \in \pi \setminus (v,u)$ s.t. $k = v$.*

Expansion of a path $\pi = [e_1, \ldots, e_i, e_{i+1}, \ldots e_n]$ where $e_1, \ldots, e_n \in E$ is another path: $\pi' = [e_1, \ldots, e_i, \ldots, e_{i+1}, \ldots e_n]$ such that the first and the last edges of π' are the same as in π, and all of the edges in π' are present in π.

Lemma 4 (Simple path expansion). *Given a CPG $G_{\mathscr{P}} = \langle N, E, L, entry, error\rangle$, and two nodes $s, t \in N$, any path π from s to t is either a simple path, or an expansion of a simple path.*

3.4.2 Extension of the Multiloop Approach to the Programs with Nested Loops

To resolve the drawbacks of the multiloop approach in verifying programs with nested loops, this section introduces a technique that extends Algorithm 1. This extension performs forward reachability analysis of the program's cutpoint graph from entry to error, gradually building a path through the system. The core difference of the extension is the analysis of nested loops, since loops can be composed of multiple nodes and edges, containing inner loops and complex branching structure. The algorithm maintains loop summaries for every loop node in the system and uses them to construct an overapproximated transition system for loop nodes. This transition system is then used to check whether the loop can be exited, and if so, to construct a path through the loop.

To keep track of the constructed loop summaries, for every node v in the CPG, the algorithm maintains a loop summary formula: Σ_v. Initially Σ_v is set to $\top$ for every node. Additionally, for every node v, the algorithm maintains the inner loop nodes of node v: $v.inner$. When the safety check is UNSAT, TPA returns the formula *reached* encoding the reached states, together with a counterexample trace ϕ_{path}. Otherwise, when the target states are unreachable, it returns a transition invariant *TrInv*. The proposed extension differs from the multiloop approach in two respects: it distinguishes between inner and outer loops, and it modifies the loop-traversal phase so that the algorithm can handle a complex inner-loop structure when a loop is nested. First, based on the DAG, the algorithm constructs all possible simple paths through the graph from the node to itself. These paths outline all possible iteration paths that can happen inside the loop. Using simple iteration paths and the loop summaries for the inner loops, the algorithm constructs a formula μ_v that overapproximates a single iteration of the loop. Based on μ_v, the algorithm constructs a Transition System and checks whether it can exit the loop using this overapproximation. TPA is used for the reachability check, allowing for a deep exploration of the loop, detecting whether an overapproximated traversal of the loop is possible. When it is, the algorithm attempts to construct the exact loop path through the CPG. In case the exact path does not exist, the inner loop summaries are refined and the algorithm traverses the loop again.

3.4.3 Core Algorithm

The main flow of Algorithm 2 is similar to Algorithm 1. For the `TraverseBridge` case, the algorithm attempts to traverse only the edges that do not lead to inner loop nodes, as the inner loop nodes are handled separately. Major differences in the algorithm are introduced in the `TraverseLoop` case, starting from Line 18, as part of

Algorithm 2: Multiloop-Nested-TPA

Input : Cutpoint Graph $G = \langle N, E, L, entry, error \rangle$;
Output: SAFE/UNSAFE

1 $path \leftarrow \{\langle \omega_{entry}, \top \rangle\}$
2 **while** $path$ *is not empty* **do**
3 **switch** *path.peek*() **do**
4 **case** $\langle \omega_v, curr \rangle$ **do**
5 **if** $\forall w \in N \setminus v.inner,\ s.t.\ (v,w) \in E : (v,w).blocked \neq \bot$ **then**
6 | SAME CODE AS IN ALGORITHM 1 | `// TraverseBridge`
7 **else**
8 | SAME CODE AS IN ALGORITHM 1 |
18 **case** $\langle \alpha_v, curr \rangle$ **do** `// TraverseLoop`
19 $\Pi_v \leftarrow$ construct all simple iteration paths from v to itself using BFS.
20 $\mu_v(X, \ldots, X') \leftarrow$ SUMMARIZEITERATION(G, v, Π_v)
21 $(res, reached, \phi_{path}, TrInv) =$ TPA$(curr(X), \mu_v(X, \ldots, X'), \neg(\omega_v.safe)(X'))$
22 **if** $res = reachable$ **then**
23 $M \models pre(X) \wedge \phi_{path}(X, \ldots, X') \wedge post(X')$
24 $loop_v \leftarrow ExtractPath(G, \phi_{path}, M, \Pi_v)$
25 **if** $\exists (w,u) \in loop_v\ s.t.\ (w,u) \neq (v,v)$ **then**
26 $refinedReached \leftarrow$ `RefineReached`$(G, v, curr, \phi_{path}, loop_v)$
27 **if** $refinedReached = \bot$ **then continue**
28 **else**
29 $refinedReached \leftarrow reached$
30 $path.push(\langle \omega_v, refinedReached \rangle)$
31 **else**
32 $\Sigma_v \leftarrow \Sigma_v \vee TrInv$
33 | SAME CODE AS IN ALGORITHM 1 |
34 **return** SAFE

handling the **Pre** case, when the algorithm enters node v with reached states *curr*. Unlike the approach in Section 3.4.3, the algorithm starts by detecting all simple paths in the CPG from node v to itself using Breadth-First Search. These paths serve as a foundation for the construction of possible paths which the loop iteration can take (in the case node v is a loop head of the nested loop). Using the detected simple paths, the algorithm constructs a formula $\mu_v(X,\dots,X') = \textsc{SummarizeIteration}(G, v, \Pi_v)$. This formula overapproximates any possible loop execution, as it uses loop summaries to overapproximate any possible execution of the nested loops. The following example illustrates the construction of the formula μ_v that overapproximates the execution of a single loop iteration.

Example 6. *Recall the CPG in Figure 3.6. Suppose the goal is to overapproximate all possible iteration paths over node s_0. There are two classes, I_1 and I_2, of iteration paths over s_0: (1) simple path $I_1 = [(s_0,s_1),(s_1,s_0)]$, which can be encoded as $L(s_0,s_1) \wedge L(s_1,s_0)$, and (2) path with inner loop execution[6] $I_2 = [(s_0,s_1)] \oplus [(s_1,s_1),\dots,(s_1,s_1)] \oplus [(s_1,s_0)]$, where $[(s_1,s_1),\dots,(s_1,s_1)]$ is some number of inner loop iterations. The path can be encoded as $L(s_0,s_1) \wedge L(s_1,s_1) \wedge \dots \wedge L(s_1,s_1) \wedge L(s_1,s_0)$. It is known that the loop summary $\Sigma_{s_1}(X,X') = x_2' \leq 33 \wedge x_1' = x_1$ overapproximates any execution of the inner loop. Therefore, any iteration path over s_1 can be overapproximated by the following formula[7]: $\mu(X^{(0)},X^{(1)},X^f,X^{(2)}) =$*

$$\overbrace{L(s_0,s_1)(X^{(0)},X^{(1)})}^{A} \wedge \overbrace{\big(\underbrace{L(s_1,s_0)(X^{(1)},X^{(2)})}_{C} \vee \underbrace{(\Sigma_{s_1}(X^{(1)},X^f) \wedge L(s_1,s_0)(X^f,X^{(2)}))}_{D}\big)}^{B}$$

Here, conjunct A captures the traversal of edge (s_0,s_1), shared between both iteration classes. Conjunct B encodes two cases: either only (s_1,s_0) is traversed without entering the loop (captured by C), or the loop over s_1 is traversed, followed by the traversal of edge (s_1,s_0) (captured by D). The formula μ overapproximates the simple path $[(s_0,s_1),(s_1,s_0)]$ or any iteration path. The concrete formula would look as follows (with variable sets $X^{(0)},X^{(2)}$ substituted by X,X'): $\mu_{s_0}(X,X^{(1)},X^f,X') =$

$$\overbrace{\big(x_1 < 512 \wedge x_1^{(1)} = x_1 \wedge x_2^{(1)} = x_2\big)}^{A} \wedge \overbrace{\Big(\big(x_2^{(1)} \geq 32 \wedge x_1' = x_1^{(1)} + x_2^{(1)} \wedge x_2' = 0\big) \vee \underbrace{\big((x_2^f \leq 33 \wedge x_1^f = x_1^{(1)}) \wedge (x_2^f \geq 32 \wedge x_1' = x_1^f + x_2^f \wedge x_2' = 0)\big)}_{D}\Big)}^{C}$$

Based on the example above, the following procedure produces formulae that overapproximate a single iteration over a specific loophead in the CPG:

Definition 14 (SummarizeIteration)**.** *Given a CPG $G = \langle N, E, L, entry, error\rangle$, a node $v \in N$, and the set Π of all simple iteration paths from node v to itself,* Summa-

[6] Here $\oplus$ denotes concatenation of paths.

[7] Throughout, X^f is a fresh copy of variables X.

RIZEITERATION *produces a formula* $\mu_v(X,\ldots,X')$[8]*:*

$$\bigvee_{\pi\in\Pi}\Big(L(e_0)(X^{(0)},X^{(1)})\wedge\bigwedge_{i=1}^{|\pi|-1}\big(\big(\Sigma_v(X^{(i)},X^f)\wedge L(e_i)(X^f,X^{(i+1)})\big)\vee L(e_i)(X^{(i)},X^{(i+1)})\big)\Big)$$

For each disjunct, the set of variables $X^{(0)}$ *is substituted by* X *and* $X^{(|\pi|)}$ *by* X'*.*

Formula μ_v overapproximates any possible single iteration from v to itself. This allows to use μ_v as a basis for the transition system, that would overapproximate loop execution over node v.

Lemma 5 (Iteration Overapproximation)**.** *Let* SUMMARIZEITERATION *over a CPG* $G=\langle N,E,L,entry,error\rangle$*, node* v*, a set of simple paths* Π_v*, return a formula* μ*. Then, for any iteration path* π_v *(path from a predicate to itself through inner loop nodes), its encoding:* $\phi(\pi_v)(X^{(0)},\ldots,X^{(n)})\implies\mu(X^{(0)},\ldots,X^{(n)})$.

Proof. The lemma is proved by contradiction. Assume there exists an iteration path π_v such that $\phi(\pi_v)(X^{(0)},\ldots,X^{(n)})\wedge\neg\mu(X^{(0)},\ldots,X^{(n)})$ is satisfiable.

By Lemma 4, any path π_v is either a simple path, or an expansion of a simple path $\pi_v=[e_1,\ldots,e_n]$. Therefore, the iteration can be written as $\pi_v=[(v,k)]\oplus\pi_k\oplus[(k,l)]\oplus\cdots\oplus\pi_r\oplus[(r,v)]$, where each loop path π_u is either an empty path or a loop path over inner loop node. Basically π_v is either a simple path, or a simple path with some loops within it.

By definition $\mu=\bigvee_{\pi\in\Pi}\mu_\pi$, where each μ_π corresponds to a specific simple path π_v. By the initial assumption, $\phi(\pi_v)\wedge\neg\mu$ is satisfiable, so $\phi(\pi_v)\wedge\neg\mu_\pi$ is also satisfiable for every μ_π. Consider specifically μ_π, which was constructed for the same simple path as the considered iteration π_v. The conjunction $\phi(\pi_v)\wedge\neg\mu_\pi$ can be written as:

$$\begin{aligned}
&\phi(\pi_v)(X^{(0)},\ldots,X^{(n)})\wedge\neg\mu_\pi(X^{(0)},\ldots,X^{(n)})=\\
&\quad(\\
&\qquad L(e_1)(X^{(0)},X^{(1)})\wedge && (e_1)\\
&\qquad\big(\phi(\pi_k)(X^{(1)},\ldots,X^{(i)})\wedge L(e_2)(X^{(i)},X^{(i+1)})\big)\wedge && (e_2)\\
&\qquad\ldots\wedge\\
&\qquad\big(\phi(\pi_r)(X^{(j)},\ldots,X^{(n-1)})\wedge L(e_n)(X^{(n-1)},X^{(n)})\big) && (e_n)\\
&)\wedge\neg(\\
&\qquad L(e_1)(X^{(0)},X^{(1)})\wedge && (e_1)\\
&\qquad\big(L(e_2)(X^{(1)},X^{(i+1)})\vee(\Sigma_k(X^{(1)},X^{(i)})\wedge L(e_2)(X^{(i)},X^{(i+1)}))\big)\wedge && (e_2)\\
&\qquad\ldots\wedge\\
&\qquad\big(L(e_n)(X^{(j)},X^{(n)})\vee(\Sigma_r(X^{(j)},X^{(n-1)})\wedge L(e_n)(X^{(n-1)},X^{(n)}))\big) && (e_n)\\
&\quad).
\end{aligned}$$

[8] X^f is an auxiliary copy of variables X, unique for each conjunct. It is needed to ensure the equivalence of the variable versions of the first and last set of variables within each conjunct. Here $|\pi|$ denotes the size of the path.

Based on the definition of the loop summaries, for any loop path π_k: $\phi(\pi_k)(X,\ldots,X') \rightarrow \Sigma_k(X,\ldots,X')$. Additionally, for any edge e_i: $L(e_i)(X,X') \rightarrow L(e_i)(X,X')$. Therefore, for each e_i: $\phi(\pi_k)(X^{(j)},\ldots,X^{(l)}) \wedge L(e_i)(X^{(l)},X^{(l+1)}) \rightarrow L(e_k)(X^{(j)},X^{(l+1)}) \vee (\Sigma_k(X^{(j)},X^{(l)}) \wedge L(e_2)(X^{(l)},X^{(l+1)}))$. This means that:

$$\phi(\pi_v) \rightarrow \mu_\pi$$

Therefore, $\phi(\pi_v) \wedge \neg\mu_\pi$ is unsatisfiable. It is a contradiction of the initial assumption that $\phi(\pi_v) \wedge \neg\mu$ is satisfiable. This completes the proof. □

After constructing a formula that overapproximates a single loop iteration over v: $\mu_v(X,\ldots,X')$ at Line 20, the algorithm proceeds to detect whether it is feasible to traverse the loop, reaching some potentially unsafe states ($\neg\omega_v.safe$). The algorithm uses TPA to check the reachability, using formula μ_v as the transition. In case potentially unsafe states are reachable, the algorithm then needs to confirm that the result was not spurious (as μ_v is an overapproximation) and to construct an exact path through the loop. When the TPA call finds a counterexample, TPA can return ϕ_{path}, which is a logical formula $\phi_{path}(X,\ldots,X') = \mu_v(X,\ldots) \wedge \cdots \wedge \mu_v(\ldots,X')$ composed of transitions, such that $curr(X) \wedge \phi_{path}(X,\ldots,X') \wedge \neg(\omega_v.safe(X'))$ is SAT. At Line 23 the algorithm extracts a variable evaluation that leads to reaching some state that satisfies $\neg\omega_v.safe$. Function ExtractPath is used to analyze this variable evaluation and to extract a path through the loop that leads to the violation of the safety property.

Definition 15 (ExtractPath). *Given a CPG system G, a satisfiable formula* $\phi = \big(pre(\ldots,X^{(0)}) \wedge \mu_v(X^{(0)},\ldots,X^{(1)}) \wedge \cdots \wedge \mu_v(X^{(n-1)},\ldots,X^{(n)}) \wedge post(X^{(n)})\big)$*, and a set of simple paths* Π_v *from v to itself, let* $M \models \phi$ *be a model of* ϕ. ExtractPath *returns the sequence of simple paths* $\{\pi_1,\ldots,\pi_n \mid M \models pre \wedge \mu_{\pi_1} \cdots \wedge \mu_{\pi_n} \wedge post$*, s.t. for all* $i \in [1,n], \pi_i \in \Pi_v\}$.

Afterwards, the algorithm checks whether a feasible path extending this sequence exists by calling `RefineReached` (Line 26), described in detail in the next section. It either constructs a concrete path through the loop, keeping track of reached states, entering and exiting every node, or returns $\emptyset$ and refines the loop summaries of the nodes used to construct μ_v.

In case when the potentially unsafe states are not reachable, the algorithm follows the same steps as Algorithm 1, updating the safe states.

3.4.4 Refinement of Reached States

This subsection describes `RefineReached`, which is called in the **Pre** stage of Algorithm 2. `RefineReached` takes as input a CPG G of a program, a node v over which the loop path is considered, formula pre, which records the set of states currently

reached in the system, a path $loop_v$ through the loop that can reach $\neg\omega_v.safe$, and a formula ϕ_{path} which overapproximates the path through the loop. The procedure either constructs reachable states that satisfy $\neg\omega_v.safe$ or, if such states do not exist, refines the loop summaries and returns $\bot$. The refined loop summaries change μ_v in Algorithm 2, blocking spurious paths and refining the overapproximation of a single loop iteration.

This procedure gradually constructs a concrete path through the CPG, maintaining the set of states that are reached within the system, following the path detected by Algorithm 2. The approach follows the same logic as Algorithm 2, restricted to a single path: it does not explore all possible paths through the cutpoint graph, but only needs to check and refine one specific path in the graph. Additionally, it keeps track of index $0 \leq id \leq 2 * |loop_v|$, defining the position in the traversed path. id is double the size of $|loop_v|$ due to the μ_v formula encoding, as iteration overapproximation encodes every edge e_i as $L(e_i)(X^{(i)}, X^{(i+1)}) \vee (L(e_i)(X^{(i)}, X^{(f)}) \wedge \Sigma_v(X^{(f)}, X^{(i+1)}))$. The algorithm traverses edges one by one and unrolls the loops encountered during traversal, producing concrete reachable states instead of the ones produced by the loop summary. In case the loop traversal does not lead to the expected set of states, it is refined using the transition invariant produced by TPA. The algorithm returns $\bot$ if the given path is spurious and the loop summaries were refined. It maintains the currently reached set of states $curr$, and updates it while traversing an exact path to reach some of the states that satisfy ϕ_{path}. The path construction follows the simplified two-case approach of Algorithm 2 in two stages, which are described below.

Post: When the algorithm exits a node, the edge traversal is simplified compared to the original approach. The procedure does not need to roll back, as if it refines some of the inner loop summaries, the reachability check of the outer loop v is executed again in Algorithm 2 to construct a different path. The algorithm approximates the states reachable after traversal of the edge (Line 6) with MBP. Formula $curr(X^{(id)}) \wedge \phi_{path}(X^{(0)}, \ldots, X^{(n)})$ is satisfiable, as initially, based on the TPA execution in Algorithm 2, $curr(X^{(0)}) \wedge \phi_{path}(X^{(0)}, \ldots, X^{(n)})$ is satisfiable, and $curr$ is updated only when it satisfies the formula $\phi_{path}(X^{(0)}, \ldots, X^{(n)})$.

Pre: When the algorithm enters a node, it attempts to traverse the corresponding loop to reach the states expected by ϕ_{path}. First, the algorithm constructs a formula representing the states expected after execution of the loop, extracting part of the formula ϕ_{path} (Line 9). Afterwards, the algorithm follows the flow of Algorithm 2 to construct the exact set of states reachable after the execution of the loop that satisfy the expected target formula. In case such traversal of the loop is impossible, the algorithm refines the loop summary (Line 21) and returns $\bot$, to refine the outer loop path and attempt the loop traversal again.

Algorithm 3: `RefineReached`$(G, v, pre, \phi_{path}, loop_v)$

Input : Cutpoint Graph $G = \langle N, E, L, entry, error\rangle$; Node $v \in N$; Preconditions formula pre; Formula which overapproximates path through the loop ϕ_{path}; Vector of traversed edges $loop_v$

Output : $refinedReached$ - refined formula of reached states, or *false* if TPA execution was spurious

1 $curr \leftarrow pre$
2 **for** $i \in (0, \ldots, |loop_v|)$ **do**
3 $id \leftarrow 2 * i$
4 $(w, u) \leftarrow loop_v[i]$
5 $M \models curr(X^{(id)}) \wedge \phi_{path}(X^{(0)}, \ldots, X^{(n)})$ `// TraverseBridge`
6 $curr \leftarrow MBP(\exists X^{(id+1)} :$ $curr(X^{(id)}) \wedge \phi_{path}(X^{(0)}, \ldots, X^{(id)}, X^{(id+1)}, \ldots, X^{(n)}), M)$
7 **if** $w \neq v$ **then**
8 **while** $\top$ **do** `// TraverseLoop`
9 $toReach(X^{(id+2)}, \ldots) \leftarrow \phi(X^{(id+2)}, \ldots)$ s.t. $\phi_{path} =$ $f(X, \ldots, X^{(id+2)}) \wedge \phi(X^{(id+2)}, \ldots)$
10 $\Pi_w \leftarrow$ construct all simple paths from w to itself using BFS.
11 $\mu_w(X, X') \leftarrow$ SummarizeIteration(G, w, Π_w)
12 $(res, reached, subpath, TrInv) =$ TPA$(curr(X), \mu_v(X, \ldots, X'), toReach(X', \ldots))$
13 **if** $res = reachable$ **then**
14 $M \models pre(X) \wedge subpath(X, \ldots, X') \wedge post(X')$
15 $subloop \leftarrow ExtractPath(G, subpath, M, \Pi_w)$
16 $refined \leftarrow$ `RefineReached`$(G, w, curr, subpath, subloop)$
17 **if** $refined = \bot$ **then continue**;
18 ;
19 $curr \leftarrow refined$
20 **else**
21 $\Sigma_w = \Sigma_w \wedge TrInv$
22 **return** $\bot$
23 **return** $curr$

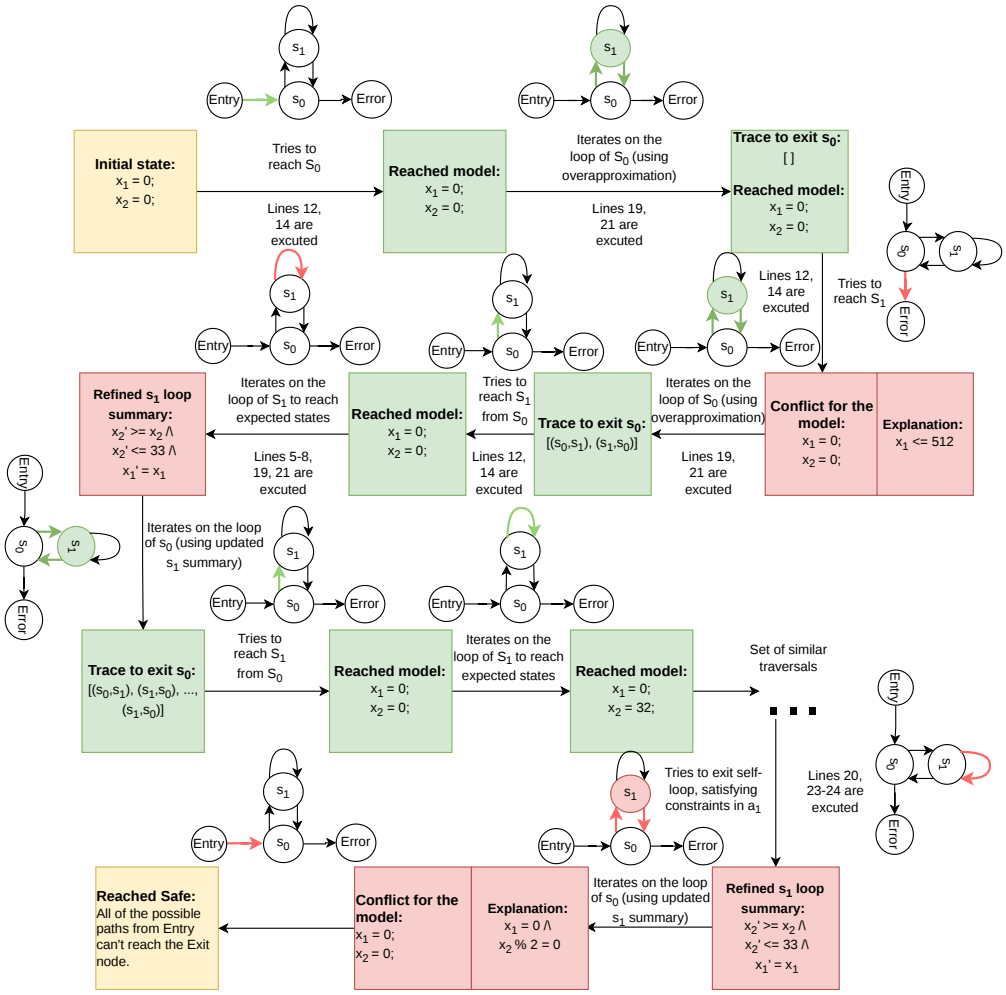


Figure 3.7. The algorithm execution flow for the example from Figure 3.6.

3.4.5 Example

To demonstrate the execution of Algorithm 2, the motivating example from Figure 3.6 is used as input. Its execution is depicted in Figure 3.7.

Initially, the algorithm execution is similar to Algorithm 1. It attempts to leave *entry* and picks the single (unblocked) outgoing edge leading to s_0. Potentially unsafe states at α_{s_0} are $\top$ at this point, so `TraverseBridge` computes reached states at α_{s_0} to be $x_1 = 0 \wedge x_2 = 0$. Attempting to continue from this state will now fail, because the only outgoing edge to *error* is not feasible, as determined by `TraverseBridge` with $x_1 \leq 512$ being the blocked states.

The algorithm now backtracks to α_{s_0} and attempts to traverse loop s_0 again, but this time with $x_1 > 512$ representing the potentially unsafe states. The transition relation is constructed as follows: $\mu_{s_0} = L(s_0,s_1)(X,X') \wedge \Sigma_{s_1}(X',X'') \wedge L(s_1,s_0)(X',X'')$. This transition relation is used in the call to TPA, which detects that potentially unsafe states are reachable in one transition and produces a corresponding path formula $\phi_{path} = \mu_{s_0}$, to which corresponds a path through the loop: $(s_0,s_1),(s_1,s_0)$. Algorithm 3 is called to construct a concrete path through the loop based on the extracted path. `RefineReached` traverses the edge (s_0,s_1), reaching the state $x_1 = 0 \wedge x_2 = 0$. Afterwards, it constructs the states that it expects to reach after the execution of the loop s_1. These states are described by the formula: $x_1' = x_1 + x_2 \wedge x_2' = 0 \wedge x_1' > 512$. These states are not reachable by the call to TPA over s_1, as it can only reach $x_2 = 32$ after one iteration. Therefore, TPA constructs a new loop summary $\Sigma_{s_1}(X,X') = x_2' \geq x_2 \wedge x_2' \leq 33 \wedge x_1' = x_1$, and `RefineReached` returns $\bot$.

Afterwards, Algorithm 2 attempts to traverse loop s_0 one more time, using the updated loop summary. Similarly to the previous attempt, it constructs μ_{s_0} with updated Σ_{s_1}, and calls TPA, which detects a potential counterexample in 16 transitions and constructs path formula $\phi_{path} = \mu_{s_0} \wedge \ldots \wedge \mu_{s_0}$ of 16 transitions, to which the corresponding path is: $[(s_0,s_1),(s_1,s_0),\ldots,(s_1,s_0)]$ — 16 iterations over loop s_0.

`RefineReached` is called over the system again, and it traverses the detected path, unrolling the loop over s_1, reaching the state $x_2 = 0 \wedge x_1 = 32$ after the first iteration, $x_2 = 0 \wedge x_1 = 64$ after the second, until it reaches the 16th iteration and it attempts to traverse loop s_1 from the state $x_2 = 0 \wedge x_1 = 480$, trying to reach $x_1' = x_1 + x_2 \wedge x_2' = 0 \wedge x_1' > 512$, which can be simplified to $x_1 + x_2 > 512$. Here TPA once again detects that these conditions are unreachable after execution of the loop over s_1 and refines the summary Σ_{s_1}, blocking the spurious counterexample, with `TraverseBridge` determining $x_1 \leq 0$ to be the safe states at ω_{s_1}.

In the final attempt to traverse s_0, TPA determines that no unsafe state is reachable anymore and computes $x_1 = 0 \wedge x_2 = 0$ as new safe states at α_{s_0}. Finally, the algorithm backtracks to ω_{entry}, and, with no new feasible way to extend the path, it concludes safety.

3.4.6 Correctness

This section first proves correctness when Algorithm 2 answers UNSAFE.

Theorem 4. *When Algorithm 2 returns* UNSAFE*, there exists a feasible path from entry to error.*

Proof. The proof follows the same induction as Theorem 2, since Algorithm 2 only changes the handling of the **Pre** case (Line 18); the **Post** case (`TraverseBridge`) is

unchanged, so the argument for entries pushed on Line 16 is identical.

For entries pushed on Line 30, a target state is first found reachable under the overapproximate transition μ_v by TPA (Line 21). This overapproximate witness is only accepted once confirmed concrete: ExtractPath (Line 23) extracts the simple paths which need to be expanded to reach the counterexample, and `RefineReached` (Line 26) attempts to confirm that simple paths correspond to an actual path through the loop which reaches the expected state. If it is confiremed with concrete path through the loops *refinedReached* is pushed onto $path$, otherwise `RefineReached` refines the inner loop summaries, updating μ_v and forces the algorithm to retry without extending $path$.

Hence, as in Theorem 2, every entry added to $path$ carries states reachable from *entry*, so Algorithm 2 returning UNSAFE (Line 17) establishes a feasible path from *entry* to *error*. □

The correctness of the SAFE answer is proved next, using some auxiliary lemmas.

Lemma 6. *The following is an invariant of the algorithm: For each node $v \in N$ and each state $s \in v.safe$ there is no feasible path from entry to error going through $\langle v, s \rangle$.*

Proof. The argument for `TraverseBridge` (Line 7) is unchanged from Lemma 1, since Algorithm 2 does not modify the **Post** case.

On Line 25, the set of safe states for node α_v is extended with the blocked states returned by `TraverseLoop`, computed with respect to the overapproximate transition μ_v rather than the exact loop transition. Since μ_v overapproximates every possible loop iteration (according to Lemma 5), any state which is unreachable under μ_v is also unreachable with the exact, concrete transitions. Thus, the blocked states remain a superset of the currently reached states in $\alpha_v.safe$ are guaranteed to only reach safe states of $\omega_v.safe$, and this extension of safe states preserves the invariant. □

Lemma 7. *When an entry $\langle v, \varphi \rangle$ is about to be popped from $path$ (Lines 8 and 26), the current path* cannot *be extended to a feasible path from entry to error.*

Proof. The proof is analogous to that of Lemma 2, using Lemma 6 in place of Lemma 1. □

Theorem 5. *When Algorithm 2 returns* SAFE*, there is no feasible path from entry to error.*

Proof. Algorithm 2 returns SAFE exactly when the initial entry $\langle entry, \top \rangle$ is popped from $path$ (Line 27). By Lemma 7, this means the current (empty) path prefix, corresponding to the unrestricted initial states $\top$, cannot be extended to a feasible path from *entry* to *error*. Hence, no feasible path from *entry* to *error* exists. □

3.4.7 Witness Production

This subsection describes witness production for Algorithm 2.

Unsafety witness. Unsafe witness production follows the procedure for Algorithm 1: the entries of $path$ form the blueprint of the witness, the traversal of every loop on the path is recorded when it succeeds, and the concrete states are obtained from a model of the loop-free formula encoding the resulting path. The difference lies in how the traversal of a loop is recorded, since a loop v with inner loops is no longer traversed by repeating a single self-loop edge.

When `TraverseLoop` succeeds for a node v (Line 30), TPA has returned a counterexample ϕ_{path} composed of k_v copies of μ_v, where k_v is the length of the trace, i.e., the number of iterations of the outer loop v. ExtractPath splits this counterexample into k_v simple iteration paths $\pi_1, \ldots, \pi_{k_v} \in \Pi_v$, one for each iteration. These paths may pass through the heads of inner loops w, whose execution μ_v only overapproximates by the loop summary Σ_w. `RefineReached` replaces each such summary by a concrete traversal: it calls TPA on the inner loop w, which again returns a counterexample whose length k_w is the number of iterations of w, and it refines the loops nested in w recursively in the same way. If `RefineReached` succeeds ($refinedReached \neq \bot$), every iteration of v and of every inner loop along the way has been made concrete. The result is a path $loop_v$ of CPG edges from v back to v, in which every inner loop w is unrolled the number of times k_w returned by TPA. The algorithm stores this path together with the entry $\langle \omega_v, refinedReached \rangle$ pushed to $path$. If v has no inner loops, $loop_v$ consists of k_v repetitions of the self-loop edge (v, v), exactly as for Algorithm 1. If `RefineReached` returns $\bot$, the spurious path is discarded, and if the algorithm backtracks over an entry, the stored path is discarded together with it.

When *error* is reached, concatenating the edges between the loops recorded in $path$ with the stored loop paths yields a concrete path through the CPG from *entry* to *error*. As before, the path is encoded as a loop-free formula, which is satisfiable because every set of states recorded during the traversal (both in $path$ and in $curr$ within `RefineReached`) contains only states that are provably reached along the path. A model of this formula gives the concrete program states of the violation witness according to Definition 9.

Safety witness. When the algorithm returns SAFE, it has proven that no feasible path exists from *entry* to *error*. The witness construction in this case is more complex compared to the original approach, as the algorithm needs to construct a witness for both the outer and inner loops. Similarly to Definition 10, inductive invariants are used as safety witnesses.

Inductive invariants can be computed from the information produced by Algorithm 2. The inductive invariant for the outer loops can be computed using the itera-

tion overapproximation μ_v and the α_v, ω_v states. Separately for each outer loop v, a safe inductive invariant is computed for a reachability problem $\langle \alpha_v.safe, \mu_v, \neg\omega_v.safe\rangle$ (which is known to be safe).[9] Afterwards, the inductive invariants for the inner loops are computed similarly. It is possible to compute all the simple paths from the outer loop head v to the inner loop head w, using DFS: Π_w. For each inner loop w, there exists a computed loop summary Σ_w. To compute pre-states for node w, the following formula should be constructed:

$$\alpha_w.safe(...,X) = \bigvee_{\pi\in\Pi_w} \big(Inv_v(X^{(0)}) \wedge \mu_\pi(X^{(0)},\ldots,X^{(n)})\big)[X^{(n)} \to X]$$

where μ_π is a formula that overapproximates the execution of the path from v to w, using the loop summaries of the inner loops.

$$\mu_\pi = L(e_0)(X^{(0)},X^{(1)}) \wedge \bigwedge_{i=1}^{|\pi|}\big((\Sigma_v(X^{(i)},X^f) \wedge L(e_i)(X^f,X^{(i+1)})) \vee L(e_i)(X^i,X^{(i+1)})\big)$$

The post-states can be computed as follows:

$$\omega_w.safe(\ldots,X) = \alpha_w.safe(\ldots,X^{(0)}) \wedge \Sigma_w(X^{(0)},X^{(1)})[X^{(1)} \to X]$$

Afterwards, the inductive invariant for each inner loop w can be computed using the reachability problem $\langle \alpha_w.safe, \mu_w, \neg\omega_w.safe\rangle$, which is guaranteed to be safe, since Σ_w overapproximates any loop traversal. Safety check can produce a safe inductive invariant for each inner loop w: Inv_w. This procedure can be applied hierarchically to all inner loops until the innermost loops are reached.

Lemma 8. *Suppose $Inv(v)$ is a safe inductive invariant for a reachability problem $\langle \alpha_v.safe, \mu_v, \neg\omega_v.safe\rangle$ for all $v \in N$. Then Inv is a safety witness according to Definition 10, i.e., $\forall(u,v) = e \in E : Inv(u) \wedge L(e) \implies Inv(v)$.*

Proof. Each $Inv(v)$ is, by construction, an inductive invariant for its corresponding loop (since μ_v is an overapproximation of a single loop iteration). For all outer loop heads the correctness of the lemma follows directly from the Lemma 3.

It remains to prove that the invariants are inductive with respect to the inner loops (and between inner loops) as well. Consider two nodes u and v, such that

[9] Any model checking algorithm can be used here, including TPA.

there exists an edge $e = (u, v)$ with $u \neq v$, and u and v are connected within an inner loop. Since $Inv(u)$ is a safe inductive invariant for the reachability problem $\langle \alpha_u.safe, \mu_u, \neg\omega_u.safe \rangle$, it follows that $Inv(u) \implies \omega_u.safe$. By construction of the algorithm, $\alpha_v.safe(\ldots, X') = \big(\omega_u.safe(\ldots, X) \wedge L(e)(X, X')\big) \vee \ldots$ The formula $\omega_u.safe \wedge L(e) \implies \alpha_v.safe$ is valid based on how the set of safe states is constructed for the inner loops. Finally, $\alpha_v.safe \implies Inv(v)$ is valid by construction of $Inv(v)$ as the inductive invariant for $\langle \alpha_v.safe, \mu_v, \neg\omega_v.safe \rangle$. All three implications together yield the desired property $Inv(u) \wedge L(e) \implies Inv(v)$.

□

3.5 Evaluation of the Extended Multiloop Technique

Algorithm 2 has been implemented in the Golem CHC solver[10]; this implementation is referred to as CSRA. It is compared to several state-of-the-art tools: Z3-Spacer (implemented in Z3, v4.13.1), Eldarica (v2.2.0), LoAT - ABMC (v0.7.0), and to existing Golem engines, namely Golem-TPA and Golem-Spacer. The evaluation focuses on complex linear CHC instances encoding programs with nested loops. Experiments were conducted on an Ubuntu 20.04 machine with an AMD EPYC 7452 32-core processor and 8x32 GiB RAM (the artefact is publicly available[11]). The evaluation addresses the following research questions:

- **RQ1:** How does the new algorithm compare to other engines in Golem?
- **RQ2:** How does CSRA perform against state-of-the-art tools?

The benchmark suite[12] is composed of nested-loop instances selected from the LIA category of CHC-COMP 2024 [70] and aeval-benchmarks [84]. Additionally, nested-loop functions were extracted from the source code of several tools: Z3, OpenSMT, SeaHorn, and Golem, and encoded as LIA CHC instances using Korn and SeaHorn. Benchmark selection is based on the presence of nested loops in the original program; for instances without nested loops, CSRA performs similarly to Golem-TPA (for example, on the LIA-lin benchmarks from CHC-Comp, which mainly contain single loops, CSRA is on average 20% faster compared to Golem-TPA and solves 20 more instances). The final set contains 130 CHC instances (68 SAT, 62 UNSAT).

Each point (x, y) in the plots represents the solving times of a particular CHC instance, with x being the execution time of CSRA and y the execution time of the

[10] https://github.com/usi-verification-and-security/golem/tree/nested-loops.
[11] https://zenodo.org/records/17055558.
[12] https://github.com/BritikovKI/pretzel-benchmarks/tree/main

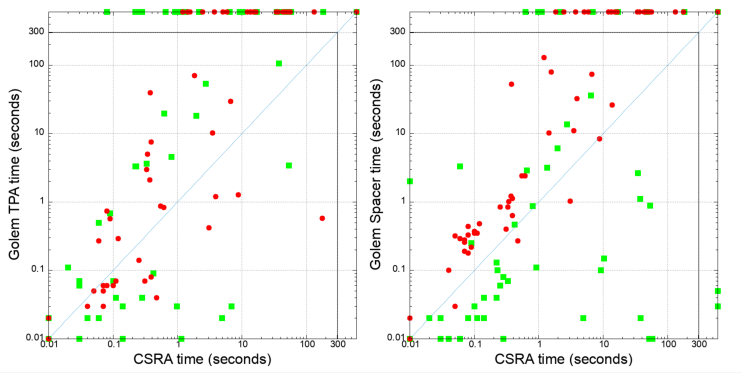


Figure 3.8. Comparison of CSRA with other engines in Golem.

competitor. Green squares are SAT instances, and red circles are UNSAT instances. Points above the diagonal indicate cases where CSRA outperformed the competitor. Points beyond 300s are timeouts for at least one of the engines.

The results indicate that CSRA consistently outperforms both Golem-Spacer and Golem-TPA for SAT and UNSAT instances. This can be attributed to the modular nature of the algorithm, which decomposes the reachability analysis task into multiple, smaller subproblems — thereby reducing the overall complexity of reasoning. CSRA leverages structural information and explicitly analyzes cycles separately, which benefits both SAT and UNSAT performance. In contrast, Golem-TPA merges the entire CHC system into a CHC cycle composed from a single clause, losing structural information. Golem-Spacer, while effective in many scenarios, struggles with deep nested loops. In particular, as the number of loop iterations increases, Golem-Spacer frequently times out. Overall, CSRA solved 118/130 instances, compared to 78 for Golem-TPA and 93 for Golem-Spacer. To answer **RQ1**, the compositional strategy of CSRA leads to better performance. This can be attributed to the *modular* summarization-based approach, which uses TPAfor fast deep reachability checks and summarization, while maintaining structural information about the system.

Figure 3.9 gives comparison charts for CSRA to other state-of-the-art CHC solvers. CSRA solves the most benchmarks (118), followed by Eldarica (111), Z3-Spacer (91), and LoAT - ABMC (55). The listed approaches are somewhat orthogonal, as Eldarica solves 6 SAT and 1 UNSAT instance uniquely, and Z3-Spacer solves 3 unique SAT problems. CSRA significantly outperforms all competitors for the UNSAT systems. In the analysis of SAT instances, CSRA is competitive with other techniques; it is only slightly outperformed by Eldarica, which solves 6 more SAT problems. To address **RQ2**, the evaluation shows that CSRA not only solves the highest number of benchmarks but also does so more efficiently. On average, for instances solved by both solvers, CSRA is 2.5× faster than Eldarica and 1.9× faster than Z3-Spacer. In summary, the pro-

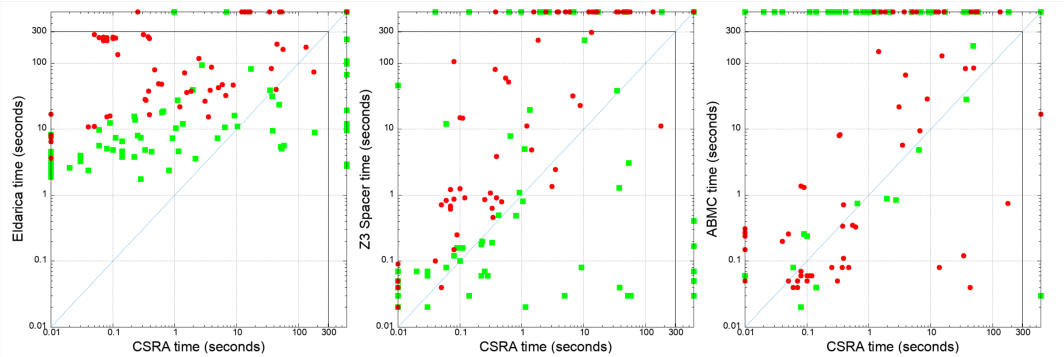


Figure 3.9. Comparison of CSRA with state-of-the-art.

posed algorithm achieves strong performance on both SAT and UNSAT CHC instances, outperforming existing GOLEM engines and other state-of-the-art solvers.

3.6 Limitations

Both approaches presented in this chapter have limitations that originate from the techniques they build on.

Theory support. Both algorithms rely on Craig interpolation and model-based projection: interpolation is used to learn blocked and safe states and inside TPA to construct transition invariants, and MBP is used to underapproximate the reached states. Consequently, they can be applied only to theories for which these operations are available. The implementation in GOLEM supports linear real and integer arithmetic (`QF_LRA` and `QF_LIA`). For example, a program containing the loop

```
while (i < n) { s = s + i * i; i = i + 1; }
```

cannot be analyzed, since the multiplication `i * i` is nonlinear, and neither can programs whose state includes arrays or other data structures.

Cost of the safety checks. Every traversal of a loop is decided by a full safety check with TPA, and a loop may be entered many times, once for every new set of reached states produced by the search. Although transition invariants are reused between these queries, a loop with a complex body, whose behavior can only be captured by complex, e.g., highly disjunctive, transition invariants, may require many refinements of TPA and dominate the overall running time. For nested loops, the cost grows further. Formula μ_v is a disjunction over all simple iteration paths through the loop body, and the number of such paths can grow exponentially with the number of branches in the body: a loop body with k consecutive `if-else` statements has up to 2^k simple iteration paths. In

addition, every spurious counterexample found with the overapproximation μ_ν triggers a refinement of the inner loop summaries and a new TPA query for the outer loop, so deeply nested loops with imprecise initial summaries may require many iterations of this refinement loop.

Extending the supported theories and reducing the cost of the safety checks, e.g., by running independent loop-level queries in parallel, are discussed in Section 3.8.

3.7 Related Work

Invariant generation. Inductive invariant generation is a widely used technique for reasoning about loops by producing an overapproximation of program states, such that any loop iteration remains within the same overapproximation. Multiple approaches include CEGAR and predicate abstraction [148], IC3/PDR [41, 115], program transformation-based methods [120], syntax-guided synthesis [83], and machine learning/neural networks [121, 162]. One of the most popular approaches, used in a wide variety of algorithms, is interpolation-based invariant generation [136]. Unlike the approaches above, the method presented in this chapter does not rely purely on state invariant generation to prove safety, but instead uses transition invariants to summarize loop behavior and reason about the system. Additionally, unlike these other techniques, the multiloop approach exploits the structural information of the system, exploring the loops in a modular and guided manner.

Lazy annotation and conflict-driven path exploration. Lazy annotation [146, 147] transfers the conflict-learning principle of CDCL SAT solvers to program verification. It searches for a feasible path to the error location in a depth-first manner, and whenever the search reaches a dead end, it uses Craig interpolation to annotate the program locations on the path with conditions that explain why the error cannot be reached from there. These annotations act like learned clauses: they prune the subsequent search, which avoids states already covered by an annotation. A related approach is Lazy Abstraction with Interpolants (LAWI) [145], which builds an abstract reachability tree labeled with interpolants and uses a covering relation between its nodes to detect when the exploration of a loop can stop. Algorithm 1 follows the same principle at the level of the cutpoint graph: it extends a path prefix ($path$), backtracks when the prefix cannot be extended, and before backtracking learns, via interpolation, sets of blocked and safe states that prevent the search from revisiting the same conflict. The main difference lies in the treatment of loops. Lazy annotation and LAWI explore loops iteration by iteration and rely on the learned annotations to eventually generalize across iterations. In contrast, Algorithm 1 treats the traversal of a whole loop as a single step of the search, decided by a TPA reachability query, and the transition invariants

it learns summarize any number of loop iterations and are reused in later queries to the same loop. This lets the search reach loop exits that require a very large number of iterations.

Decomposition and distributed verification. Another line of work decomposes a verification task along the structure of the program. Spacer [124] computes under- and overapproximating summaries of procedures for the verification of recursive programs. Beyer et al. [28] decompose a verification task into multiple smaller, connected tasks, one for each block of the program's control flow, and compute block summaries by distributed, continuous refinement. For every block, preconditions are synthesized at the block entry from the postconditions of its predecessors, and violation conditions are synthesized at the block exit from the violation conditions of its successors. Since every block is an independent verification problem, all blocks can be analyzed in parallel, and the analysis of a block is restarted whenever new information from its neighbors becomes available. The approaches in this chapter share the idea of decomposing the system along its structure and of exchanging information about states between its components: the reached states play the role of preconditions passed forward to the next loop, and the safe states $\alpha_v.\mathit{safe}$ and $\omega_v.\mathit{safe}$ play the role of (negated) violation conditions passed backward. The decomposition, however, serves a different purpose. The approach of Beyer et al. is designed for distribution, analyzing all blocks independently and in parallel, whereas the approaches in this chapter use loops as the unit of decomposition and are guided by a single path search, so that information is computed only for the states actually reached along the current path. Running the independent loop-level queries in parallel, in the spirit of distributed summary synthesis, is a natural direction for combining the two ideas.

Single-loop transformation. A common strategy for analyzing complex systems is to transform them into a single-loop representation [1, 75]. This enables techniques such as IMC or TPA, which are optimized for single-loop analysis [29], to be applied to the original, more complex program structure. This transformation, however, leads to a loss of structural information, which can negatively impact the performance of the analysis.

Loop acceleration. Techniques [15, 40] aim to produce quantifier-free first-order formulae that are equivalent to or underapproximate a loop's behavior. These methods are motivated by improving the scalability of safety verification, attempting to replace a loop with a concrete formula. State-of-the-art acceleration approaches predominantly rely on underapproximation and can struggle with proving safety. Unlike the presented technique, state-of-the-art loop acceleration techniques are limited to simple loops and rely on underapproximation, which makes them unsuitable for proving safety.

Loop summarization. A group of approaches overapproximates loop execution by constructing a symbolic relation between loop inputs and outputs, allowing the

loop to be abstracted away in subsequent analysis. Common methods include symbolic abstraction [37, 126, 168] and symbolic execution [101, 169]. However, such approaches are typically property-agnostic and unaware of the surrounding system, which can reduce their effectiveness. Some of the approaches above also do not support summarization of nested loops. In contrast, the method presented here performs property-guided summary generation and handles nested loops.

3.8 Future Work

The techniques presented in this chapter enable modular, efficient safety verification of software systems with multiple and nested loops, while taking the structure of the verified system into account. Nevertheless, several extensions and improvements to this work are possible.

A direct way to address the theory limitation discussed in Section 3.6 is to extend the approach to further theories, such as arrays or nonlinear arithmetic. The algorithms themselves do not depend on a particular theory; they only require interpolation and model-based projection, so they can be applied to any theory for which the SMT solver supports these operations.

One such extension is parallelized safety verification. Due to the modular nature of both algorithms, the analysis can be parallelized, exploring multiple paths through the CPG simultaneously. The parallel threads could share information as it is discovered, jointly updating the transition invariants and the incoming and outgoing state sets associated with each CPG node. This would significantly speed up the analysis, both when proving safety and when searching for counterexamples.

Another direction is to reverse the search and adopt a PDR-like, backward exploration. The algorithm would start from the unsafe states and explore backward toward the initial states of the system. This backward search could be combined with the existing forward analysis and potentially improve verification performance, especially when the two directions are run in parallel.

A further direction is to use the approach for incremental verification. During analysis, many artifacts are produced: the explored states of each loop, transition invariants, and state invariants, all of which carry information useful for verification. If only part of the program is subsequently changed, the artifacts already computed for the unaffected parts can be reused, and only the parts invalidated by the change need to be recomputed, significantly speeding up verification. This would make the technique far better suited to integration into an active development workflow, such as regression verification or continuous integration.

3.9 Conclusion

This chapter introduced two approaches: one for safety verification of programs with multiple loops, and its extended approach for the verification of programs with nested loops. The idea that connects both approaches is a modular analysis and path-by-path exploration of the system. The multiloop approach propagates information about reachable and blocked states between consecutive loops. The utilization of Transition Power Abstraction for the analysis of individual loops enables incremental computation and use of transition invariants for program verification, which significantly improves the overall performance of the approach. The extended technique additionally exploits the capability of TPA to produce transition invariants to the full capacity. Produced transition invariants serve as a loop summary, offering efficient and compact over-approximations of nested loop systems. The chapter proved the correctness of both algorithms and demonstrated how witnesses, both for SAFE and UNSAFE instances, can be generated. The experimental evaluation demonstrates that the proposed approaches significantly outperform a straightforward application of TPA as well as other competitors in the analysis of multiloop and nested-loop systems. Two novel solutions have been presented at FM 2024 and iFM 2025, with [47] winning the distinguished paper award at FM 2024.

Chapter 4

Mutually Guided Termination and Nontermination Analysis

Termination analysis of infinite-state systems is a fundamental problem in formal methods. Proving termination and nontermination is undecidable in general, yet there exist many techniques that succeed on a large class of real-world problems. Modern techniques treat termination and nontermination as separate problems, since each has a different proof procedure. The existence of a ranking function by e.g. [21, 85, 96, 109, 128, 156, 160] witnesses termination by providing a measure that strictly decreases along every transition. Alternatively, fixpoint computation, e.g. [173], and disjunctively well-founded transition invariants [27, 66, 127, 171] have been proposed for termination analysis. Nontermination techniques are predominantly based on the detection of recurrent sets — sets of states from which execution cannot escape [52]. There also exist alternative approaches based on Hoare-style reasoning [133], safety proving, representing infinite runs as sums of geometric series [134], or synthesis of *negative* exact loop bounds.

This chapter presents a unified framework for proving both termination and nontermination. This framework extends the safety-based nontermination analysis (SNA), enabling it to prove termination. The extension is a novel interpolation-based method for generating disjunctively well-founded transition invariants for termination analysis. The proposed approach uses terminating transition traces constructed by SNA, overapproximating them with a Craig interpolation. The interpolant is processed into a transition invariant candidate, which is used to prove termination. Additionally, based on the transition invariant candidate, the algorithm creates smaller, focused (non)termination problems, which guide the analysis of the whole transition system. Solving these problems helps to extend the transition invariant candidate and enables an efficient nontermination analysis of complex problems.

The new technique was implemented in the Golem Constrained Horn Clause solver and evaluated on the Integer Transition Systems benchmarks from the Termination Competition [97]. Our tool solved a large number of terminating instances and demonstrated significant improvement for nontermination analysis, compared to the classical SNA. The comparison with the winners of the Termination Competition 2025[1]—KoAT, LoAT, and T2— shows that the approach is competitive with state-of-the-art tools and can solve unique instances, including those that were never solved by any tool before.

Overall, the contributions of this chapter are as follows: (1) a novel interpolation-based method for the generation of disjunctively well-founded transition invariants, (2) a unified termination/nontermination analysis procedure that utilizes the intermediate results of (non)termination analysis for self-guidance, and (3) its implementation and evaluation. This work is going to be submitted to TACAS 2027 conference.

4.1 Preliminaries

This section introduces the required definitions and formalisms required for the introduced termination analysis framework.

This contribution is based on the Transition System $TS = \langle Init, Tr, X \rangle$ and safety problem $P = \langle Init, Tr, Bad, X \rangle$ formalisms, introduced in Chapter 2. It also relies on the notions of termination and nontermination, recurrent sets, ranking functions, and disjunctively well-founded transition invariants introduced in Section 2.2.4.

SafetyCheck (P) denotes a safety verification procedure. If P is safe, it returns a tuple $\langle \text{SAFE}, Inv(X) \rangle$, where $Inv(X)$ is an inductive invariant. If P is unsafe, it returns $\langle \text{UNSAFE}, Tr^n(X^{(0)}, \ldots, X^{(n)}) \rangle$, a counterexample trace for which the following formula is satisfiable:

$$Init(X^{(0)}) \wedge Tr^n(X^{(0)}, \ldots, X^{(n)}) \wedge Bad(X^{(n)})$$

The SafetyCheck function can be instantiated with any automated safety verification procedure, such as TPA, Spacer, or the multiloop procedure described in Chapter 3.

4.1.1 Proving Nontermination via Safety

Many techniques reduce termination and nontermination analysis to safety verification. One notable approach is the reduction of nontermination to safety, which constructs recurrent sets via inductive invariants. It was introduced in the context of proving nontermination of programs with loops, and can be extended to transition systems.

[1] https://termination-portal.org/wiki/Termination_Competition_2025.

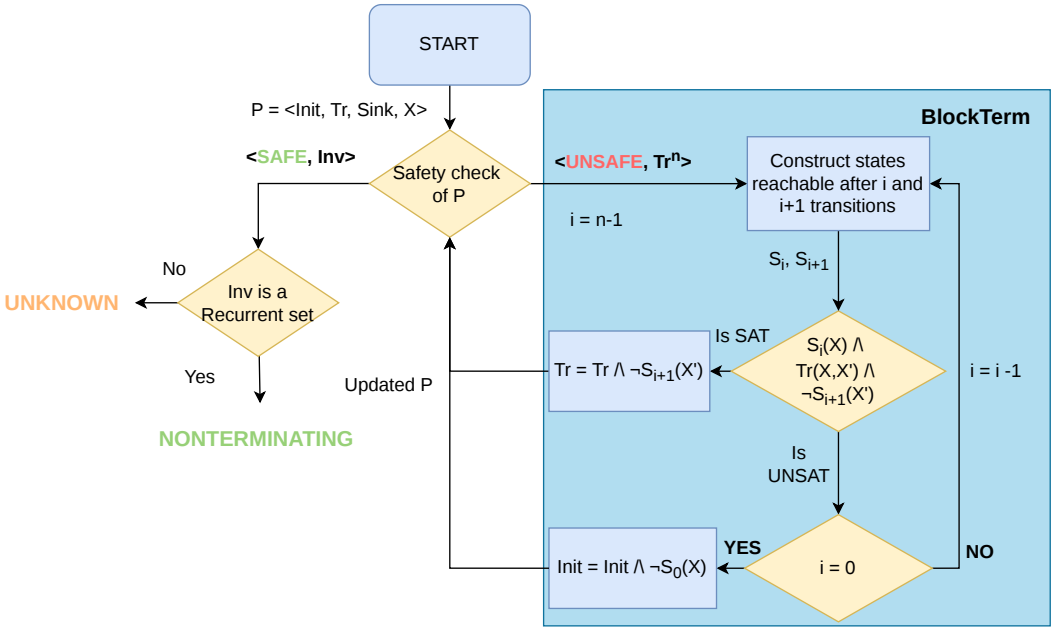


Figure 4.1. Execution flow of Safety-based Nontermination analysis.

Definition 16 (Sink states)**.** *For a Transition System* $\langle Init, Tr, X\rangle$*, a state* s *is a sink state, if* $\neg\exists s'.Tr(s,s')$*.* $Sink(X)$ *denotes the formula encoding all sink states in the transition system:* $\forall X.Sink(X) \leftrightarrow \neg\exists X'.Tr(X,X')$

Algorithm 4 for Safety-based Nontermination analysis (SNA) takes a safety problem $P = \langle Init, Tr, Sink, X\rangle$, where sink states are constructed from TS using quantifier elimination (QE): $Sink(X) = \neg\text{QE}(\exists X'.Tr(X,X'))$. As an output, it returns NONTERM if the system is nonterminating, and UNKNOWN, if the algorithm cannot conclude the analysis. Figure 4.1 illustrates the execution flow of Safety-based Nontermination.

SNA iteratively solves increasingly refined safety problems until the given sink states are proven unreachable. In every iteration, if there exists a trace reaching sink states (line 3), BlockTerm (Algorithm 5) refines the safety problem P by blocking the states that deterministically lead to the sink states, updating the safety problem. BlockTerm is described in detail below. When the original sink states are no longer reachable, SNA checks if the computed inductive invariant $Inv(X)$ is a recurrent set (line 6). This amounts to deciding satisfiability of the following formula:

$$Inv(X) \wedge \neg\exists X'.Tr(X,X').$$

If $Inv(X)$ is a recurrent set, then the transition system is nonterminating. Otherwise, SNA returns UNKNOWN.

Algorithm 4: Safety-based Nontermination Analysis (SNA).

Input : Safety problem $P = \langle Init, Tr, Sink, X \rangle$
Output: NONTERM | UNKNOWN

```
1 while ⊤ do
2 |  ⟨res, φ = Tr^n or Inv⟩ ← SafetyCheck(P)
3 |  if res = UNSAFE then
4 |  |  P ← BlockTerm(P, φ)
5 |  else
6 |  |  if ∀X.φ(X) → ∃X'.Tr(X, X') then
7 |  |  |  return NONTERM
8 |  |  return UNKNOWN
```

Algorithm 5: BlockTerm (P, Tr^n).

Input : Safety problem $P = \langle Init, Tr, Sink, X \rangle$, terminating trace $Tr^n(X^{(0)}, \ldots, X^{(n)}) = Tr(X^{(0)}, X^{(1)}) \wedge \cdots \wedge Tr(X^{(n-1)}, X^{(n)})$
Output: Refined safety problem P'
Data : $S_0, \ldots, S_n$ - formulae which encode states within trace reachable in 0, ..., n transitions

```
1  i ← n − 1
2  while i > 0 do
3  |  S_{i+1}(X) ← QE(∃X^(0), ..., X^(i), X^(i+2), ..., X^(n).
   |    Init(X^(0)) ∧ Tr^n ∧ Sink(X^(n)))[X^(i+1) ↦ X]
4  |  S_i(X) ← QE(∃X^(0), ..., X^(i−1), X^(i+1), ..., X^(n).
   |    Init(X^(0)) ∧ Tr^n ∧ Sink(X^(n)))[X^(i) ↦ X]
5  |  if SAT ? S_i(X) ∧ Tr(X, X') ∧ ¬S_{i+1}(X') then
6  |  |  Tr'(X, X') ← Tr(X, X') ∧ ¬S_{i+1}(X')
7  |  |  return ⟨Init, Tr', Sink, X⟩
8  |  i ← i − 1
9  Init'(X^(0)) ← Init(X^(0)) ∧ ¬QE(∃X^(1), ..., X^(n).Init(X^(0)) ∧ Tr^n ∧ Sink(X^(n)))
10 return ⟨Init', Tr, Sink, X⟩
```

The BlockTerm procedure takes as input a safety problem P and a trace formula $Tr^n(X^{(0)}, \ldots, X^{(n)})$ such that $Init(X^{(0)}) \wedge Tr^n(X^{(0)}, \ldots, X^{(n)}) \wedge Sink(X^{(n)})$ is satisfiable. The algorithm iteratively analyzes the trace from the last transition backward. Using QE, Algorithm 5 computes S_i and S_{i+1}, encoding the states that are reachable after i and $i+1$ transitions from $Init$ (lines 3, 4). Then, at line 5 it checks if $S_i(X) \wedge Tr(X, X') \wedge \neg S_{i+1}(X')$ is satisfiable. Intuitively, with this query, the algorithm checks if it is possible to take an alternative transition at step i, one that does not lead to the sink states. If it is possible, the algorithm blocks the states deterministically leading to the sink states by

excluding them from the transition relation (line 6). If the whole trace is deterministic (it is not possible to take an alternative transition along the trace), BLOCKTERM blocks the subset of initial states deterministically leading to sink (line 9).

Lemma 9. *If Algorithm 4 returns* NONTERM*, then the transition system is nonterminating.*

Example 7. *Consider a transition system* $TS = \langle Init, Tr, X \rangle$ *with* $X = \{x\}$, $Init(X) \triangleq \top$, *and* $Tr(X, X') \triangleq (x' = x-1 \vee x' = x-2) \wedge x \neq 0$. *The sink states are* $Sink(X) \triangleq x = 0$, *and the initial safety problem is* $P = \langle Init, Tr, Sink, X \rangle$. *This* TS *is nonterminating, since* x *infinitely decreases.*

Algorithm 4 updates the initial states and the transition relation in two iterations, via BLOCKTERM*:* $Init(X) \leftarrow x \neq 0$, *and* $Tr(X, X') \leftarrow Tr(X, X') \wedge x' \neq 0$. *The safety check of the updated problem concludes* SAFE*, with* $Inv(X) \triangleq x \neq 0$. $Inv(X)$ *is a recurrent set, and algorithm returns* NONTERM.

Motivation for transition invariant-based termination analysis. Consider a small modification (replacing $x \neq 0$ *with* $x > 0$*):* $Tr(X, X') \triangleq (x' = x-1 \vee x' = x-2) \wedge x > 0$. *This renders* TS *terminating since* x *is strictly positive and decreases every step. However, Algorithm 4 cannot detect termination by design, and it would return* UNKNOWN.

Consider the execution of Algorithm 4 on this example. First, using BLOCKTERM*, it blocks initial states that instantly terminate:* $Init(X) \leftarrow x > 0$. *In the second iteration,* SAFETYCHECK (P) *returns* UNSAFE *with a trace* $Tr^1(X^{(0)}, X^{(1)})$ *satisfying* $\exists X^{(0)}, X^{(1)}. Init(X^{(0)}) \wedge Tr^1(X^{(0)}, X^{(1)}) \wedge Sink(X^{(1)})$. *The key insight is to generalize terminating traces into a formula describing an entire class of terminating behaviors. If this generalization yields a disjunctively well-founded transition invariant over all reachable states, it witnesses termination. Particularly,* $TrInv(X, X') \triangleq x' < x \wedge x > 0 \wedge x' > -2$ *is a well-founded transition invariant that proves the termination of* TS. *Section 4.2 presents the algorithm that automates this generalization via Craig interpolation.*

4.2 Termination Analysis via Interpolation-based Transition Invariant Generation

This section presents a new termination analysis approach that leverages safety verification to generate a well-founded transition invariant via interpolation. At each iteration, the procedure constructs a trace from the initial states to a sink state. A candidate well-founded transition invariant is then constructed based on the overapproximation of the terminating trace. If the candidate is valid for all reachable states (meaning it is a transition invariant), termination is concluded. The interpolation-based approach enables efficient generation of a transition invariant guided by the termination conditions.

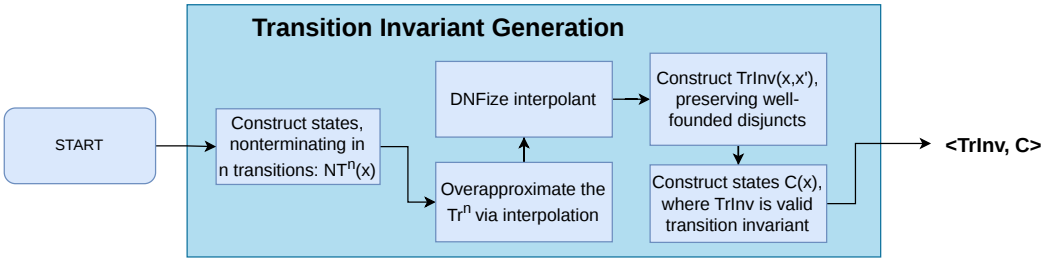


Figure 4.2. Execution Flow of Interpolation-based Transition Invariant Generation

4.2.1 Interpolation-based Generation of Well-Founded Transition Invariants

The core of the approach is TrInvGen, a procedure for the automated interpolation-based generation of well-founded transition invariants. Given a transition system and a terminating trace, it constructs a disjunctively well-founded approximation of the trace. If the resulting formula constitutes a transition invariant over all reachable states, it witnesses termination.

The procedure consists of five steps: (1) Computing NT^n, the formula encoding states from which non-sink states can be reached within n transitions. NT^n is used to construct the interpolant. The negation of NT^n encodes the states from which *Sink* is guaranteed to be reached in at most n transitions. (2) Constructing an overapproximation of the terminating trace via interpolation. The usage of sink states allows the interpolant to capture the reason for termination. (3) Converting the interpolant into an underapproximating DNF formula. This enables efficient ranking function synthesis for each disjunct. (4) Filtering disjuncts, preserving only disjuncts that encode well-founded relations. These disjuncts are used to construct a candidate transition invariant. Only well-founded relations are preserved, as the transition invariant needs to be disjunctively well-founded to witness termination. (5) Computing the formula $C(X)$, encoding states for which the candidate transition invariant is valid. The transition invariant needs to be valid only over the reachable states to conclude termination. Therefore, if all reachable states satisfy $C(X)$, then TS is terminating. The overall flow of the procedure is illustrated in Figure 4.2 and described in Algorithm 6.

Algorithm 6 takes as input a safety problem $P = \langle Init, Tr, Sink, X\rangle$, a transition trace $Tr^n(X^{(0)}, X^{(1)}, \ldots, X^{(n)})$, and a transition invariant formula $TrInv(X, X')$. It returns a tuple of formulae $\langle TrInv, C\rangle$, where $TrInv(X, X')$ is a disjunctively well-founded transition invariant candidate, and $C(X)$ is a formula encoding the states for which *TrInv* is a valid transition invariant. It proceeds in the following steps:

Step 1: *Computing NT^n (line 1).* $NT^n(X)$ encodes all states from which a non-sink state is reachable within n transitions. It is defined by the following property, for all

Algorithm 6: TrInvGen$(P, Tr^n, TrInv)$.

Input : Safety problem $P = \langle Init, Tr, Sink, X\rangle$, transition trace $Tr^n(X^{(0)}, X^{(1)}, \ldots, X^{(n)})$, candidate transition invariant $TrInv(X, X')$

Output: Updated transition invariant $TrInv(X, X')$ and formula $C(X)$ encoding states covered by transition invariant

1 $NT^n(X^{(0)}) \leftarrow \text{QE}(\exists X^{(1)}, \ldots, X^{(n)}.\ Tr^n(X^{(0)}, X^{(1)}, \ldots, X^{(n)}) \wedge \neg Sink(X^{(n)}))$
2 $itp(X, X') \leftarrow \text{Itp}(Tr^n(X, \ldots, X'), \neg NT^n(X) \wedge \neg Sink(X'))$
3 $Cs(X, X') \leftarrow \text{ToDNF}(itp)$
4 $TrInv \leftarrow TrInv \vee \bigvee_i \{d_i \in Cs \mid \exists \text{ ranking function } f(X)\}$
5 $C(X) \leftarrow \neg\text{QE}(\exists X', X''.\ \big(Id(X, X') \vee TrInv(X, X')\big) \wedge Tr(X', X'') \wedge \neg TrInv(X, X''))$
6 **return** $\langle TrInv, C\rangle$

$X^{(0)}$:

$$NT^n(X^{(0)}) \leftrightarrow \exists X^{(1)}, \ldots, X^{(n)}.Tr^n(X^{(0)}, \ldots, X^{(n)}) \wedge \neg Sink(X^{(n)})$$

NT^n is constructed with quantifier elimination, eliminating all variables from the formula except for $X^{(0)}$. The construction of NT^n enables the overapproximation of the trace.

Example 8. *Consider* $TS = \langle Init, Tr, X\rangle$ *with* $Init \triangleq x > 0$, $Tr(X, X') \triangleq (x' = x - 1 \vee x' = x - 2) \wedge (x > 0 \vee x < -10)$, $X \triangleq \{x\}$. *The sink states are* $Sink(X) \triangleq x \leq 0 \wedge x \geq -10$. *For* $n = 1$, *quantifier elimination yields* $NT^1(X) = x > 1 \vee x < -10$*: states from which* $\neg Sink$ *is reachable in one transition.*

Step 2: *Interpolation (line 2).* The following formula is unsatisfiable by construction since any state that satisfies $\neg NT^n$ must reach *Sink* within at most n steps:

$$\neg NT^n(X^{(0)}) \wedge Tr^n(X^{(0)}, \ldots, X^{(n)}) \wedge \neg Sink(X^{(n)})$$

This allows computing a *Craig interpolant* that overapproximates the trace Tr^n. Crucially, the usage of the sink states for the interpolant construction allows the interpolant to capture the reason *why* the particular trace is terminating.

Step 3: *DNF underapproximation (line 3).* For a transition invariant to witness termination, it must be disjunctively well-founded: every disjunct must encode a well-founded relation. Converting the interpolant to DNF would simplify the well-foundedness check for separate disjuncts. However, complete DNF conversion can cause an exponential blowup in formula size and take a significant amount of time. Moreover, even a complete DNF is not guaranteed to yield a transition invariant.

Instead, the algorithm follows a lazy approach: it constructs an *underapproximating* DNF formula on demand. Specifically, the algorithm constructs disjuncts as follows:

(1) the SMT solver produces a model that satisfies itp; (2) the literals satisfied by this model are extracted and conjoined to form a single disjunct; (3) this disjunct is blocked from itp and the process repeats until some m disjuncts are constructed. The result is a formula:

$$Cs(X,X') \triangleq \bigvee_{i=0}^{m} d_i$$

where each d_i is a conjunction, and $Cs(X,X') \rightarrow itp(X,X')$. This approach significantly reduces formula size and speeds up both the DNF construction and subsequent well-foundedness check, while preserving correctness: in the worst case, *Cs* fails to be a transition invariant, but this is also possible even with a complete DNF conversion.

Step 4: *Filtering Well-Founded Disjuncts (line 4).* For each disjunct $d(X,X') \in Cs(X,X')$, the algorithm checks if $d(X,X')$ encodes a well-founded relation. The existence of a ranking function is checked by automatically synthesizing a linear ranking function. The disjuncts admitting a ranking function are preserved, the rest are discarded. The resulting formula is disjoined with the input candidate transition invariant *TrInv*.

Step 5: *Synthesis of the states for which the transition invariant is valid (line 5).* If *TrInv* is a transition invariant over all reachable states, then TS terminates, since every disjunct encodes a well-founded relation. To verify this, the procedure computes the formula $C(X)$ encoding the states over which *TrInv* is a valid transition invariant (including sink states, as any candidate invariant is valid over *Sink*).

Definition 17 (Coverage formula)**.** *Given a transition system $TS = \langle Init, Tr, X\rangle$ and a formula $TrInv(X,X')$, a formula $C(X)$ is called a coverage formula if the following holds for all X, X', X'':*

$$C(X) \wedge Tr(X,X') \rightarrow TrInv(X,X')$$
$$C(X) \wedge TrInv(X,X') \wedge Tr(X',X'') \rightarrow TrInv(X,X'')$$

That is, TrInv is a transition invariant over states that satisfy $C(X)$.

Similarly to NT^n, $C(X)$ is constructed by applying quantifier elimination, negating the states for which *TrInv* is not a transition invariant. It is a coverage formula by construction.

Example 9. *Continuing Example 8, suppose the procedure constructs an interpolant $itp(X,X') \triangleq x' < x \wedge ((x > 0 \wedge x' > -2) \vee x < -10)$. Afterwards, the algorithm would construct the DNF form: $(x' < x \wedge x > 0 \wedge x' > -2) \vee (x' < x \wedge x < -10)$, and since the second disjunct is not well-founded, it would be dropped, resulting in $TrInv(X,X') \triangleq x' < x \wedge x > 0 \wedge x' > -2$. Quantifier elimination at line 5 then returns $C(X) \triangleq x \geq -10$.*

Note. If $\neg C(X)$ is unreachable, then $TrInv(X,X')$ is a disjunctively well-founded transition invariant over all reachable states, and witnesses termination of the transition system. For illustration, the transition invariant in Example 9 is sufficient to prove termination since the states satisfying $\neg C(X) \triangleq x < -10$ cannot be reached within TS.

Lemma 10. *Let $TS = \langle Init, Tr, X\rangle$ be a transition system and let $C(X)$ be a formula such that $\neg C(X)$ is unreachable in TS. If $TrInv(X,X')$ is a finite disjunction of formulae encoding well-founded relations and satisfies $\forall X, X', X''$:*

$$C(X) \wedge Tr(X,X') \rightarrow TrInv(X,X')$$
$$C(X) \wedge TrInv(X,X') \wedge Tr(X',X'') \rightarrow TrInv(X,X'')$$

then TS is terminating.

Proof. The proof shows that $TrInv$ is a disjunctively well-founded transition invariant over the reachable states of TS, from which termination follows.

Disjunctive well-foundedness. By construction, for each disjunct d_i of $TrInv = \bigvee_i d_i$ there exists a ranking function. Therefore, every disjunct is a logical encoding of well-founded relation. Hence $TrInv$ is disjunctively well-founded.

Transition invariant. It remains to show that $\forall X, X'. R(X) \wedge Tr^+(X,X') \rightarrow TrInv(X,X')$, where $R(X)$ denotes the reachable states of TS, by induction on the number of steps n in $Tr^+(X,X')$.

Base case ($n = 1$): It is needed to prove that:

$$R(X) \wedge Tr(X,X') \rightarrow TrInv(X,X')$$

Since $\neg C(X)$ is unreachable, $R(X) \rightarrow C(X)$, and therefore $R(X) \wedge Tr(X,X') \rightarrow C(X) \wedge Tr(X,X')$.

The following assumption holds by the lemma statement:

$$C(X) \wedge Tr(X,X') \rightarrow TrInv(X,X').$$

Therefore the base case holds.

Inductive step: Assume $R(X) \wedge Tr^n(X,\ldots,X') \rightarrow TrInv(X,X')$ for some $n \geq 1$. Let $Tr^{n+1}(X,X'')$ hold, so that $\exists X,\ldots,X',X''.R(X) \wedge Tr^n(X,\ldots,X') \wedge Tr(X',X'')$. Since $R(X) \wedge Tr^n(X,\ldots,X') \rightarrow TrInv(X,X')$, then trivially $R(X) \wedge Tr^n(X,\ldots,X') \wedge Tr(X',X'') \rightarrow R(X) \wedge TrInv(X,X') \wedge Tr(X',X'')$. Now, due to the fact that $R(X) \rightarrow C(X)$, the following holds: $R(X) \wedge TrInv(X,X') \wedge Tr(X',X'') \rightarrow C(X) \wedge TrInv(X,X') \wedge Tr(X',X'')$. By the lemma hypothesis:

$$C(X) \wedge TrInv(X,X') \wedge Tr(X',X'') \rightarrow TrInv(X,X'').$$

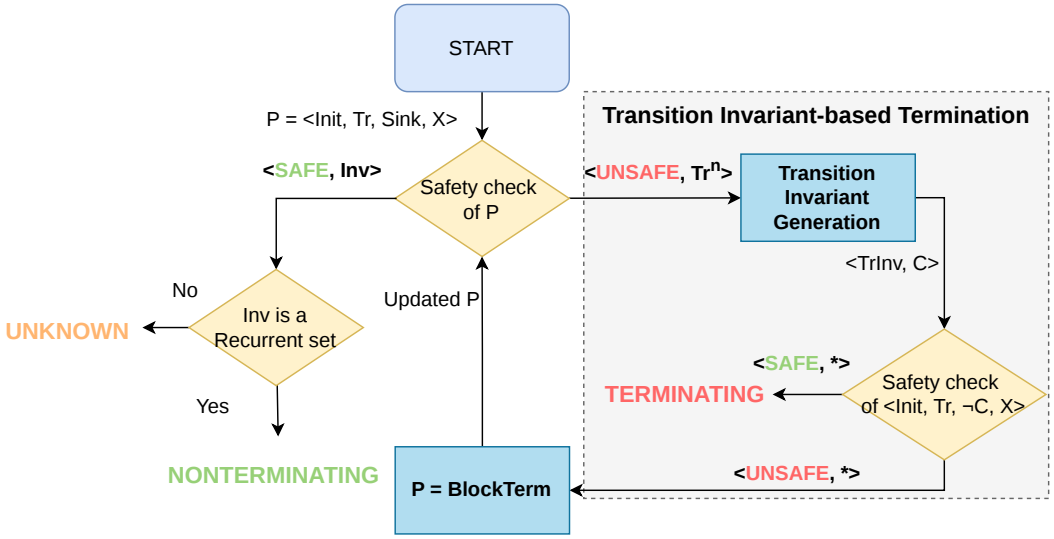


Figure 4.3. Execution Flow of Interpolation-based Transition Invariant Generation (new components are placed in a gray block, lines 5,6,8).

Therefore, through the chain of implications:

$$R(X) \wedge Tr^{n+1}(X, X'') \rightarrow TrInv(X, X'').$$

Thus *TrInv* is a disjunctively well-founded transition invariant over all reachable states, and TS is terminating. □

4.2.2 Interpolation-based Termination Analysis

This subsection presents the full termination analysis procedure using the interpolation-based transition invariant generation. It integrates TrInvGen into the main loop of Algorithm 4. At each iteration, instead of only blocking terminating states, the algorithm also constructs a transition invariant candidate. The overall flow is shown in Figure 4.3.

Algorithm 7 takes as input a safety problem $P = \langle Init, Tr, Sink, X \rangle$, where *Init* and *Tr* encode the initial states and transition relation, *Sink* is the formula encoding sink states, and X is the set of variables over which the transition system is defined. As an output, the algorithm returns NONTERM if TS is nonterminating, TERM if it is terminating, and UNKNOWN if the procedure cannot determine the final result. The procedure maintains a candidate transition invariant $TrInv(X, X')$. During the execution, *TrInv* is extended with new disjuncts, which are generated by the TrInvGen procedure, and encode well-founded relations. Changes to the original SNA are at lines 5, 6, and 8.

Algorithm 7: Interpolation-based Transition Invariant Generation for Proving and Disproving Termination.

Input : Safety problem $P = \langle Init, Tr, Sink, X \rangle$
Output: TERM | NONTERM | UNKNOWN
Data : *TrInv* - formula that encodes transition invariant

```
1  TrInv ← ⊥
2  while ⊤ do
3      ⟨res, φ = Tr^n or Inv⟩ ← SafetyCheck(P)
4      if res = UNSAFE then
5          ⟨TrInv, C⟩ ← TrInvGen(P, φ, TrInv)
6          ⟨res', _⟩ ← SafetyCheck(⟨Init, Tr, ¬C, X⟩)
7          if res' = UNSAFE then P ← BlockTerm(P, φ)
8          else return TERM
9      else
10         if ∀X.φ(X) → ∃X'.Tr(X, X') ∧ φ(X') then
11             return NONTERM
12         return UNKNOWN
```

Every iteration of Algorithm 7, similarly to Algorithm 4, begins with the safety check at line 3. The algorithm handles two possible cases:

UNSAFE A terminating trace Tr^n exists. The algorithm calls TrInvGen to extend *TrInv* and compute the coverage formula $C(X)$ (line 5). Then, it checks if $\neg C(X)$ is reachable from the initial states (line 6). If $\neg C(X)$ is reachable, the algorithm refines the safety problem by blocking the states that deterministically lead to termination via BlockTerm, and the loop continues. Otherwise, *TrInv* is a transition invariant over all reachable states, and the algorithm concludes termination.

SAFE No sink state is reachable, and there exists an inductive invariant $Inv(X)$. If $Inv(X)$ is a recurrent set, the system is nonterminating. Otherwise, the algorithm returns UNKNOWN as the analysis is inconclusive.

Example 10. *Consider TS from Example 8 with* $Init \triangleq x > 0$, $Tr \triangleq (x' = x-1 \vee x' = x-2) \wedge (x > 0 \vee x < -10)$, $X \triangleq \{x\}$. *Preprocessing computes* $Sink \triangleq x \leq 0 \wedge x \geq -10$. *Then, the algorithm is called with* $P = \langle Init, Tr, Sink, X \rangle$.

The safety check detects a trace Tr^1 *reaching Sink.* TrInvGen *produces a transition invariant candidate* $TrInv(X, X') \triangleq x' < x \wedge x > 0 \wedge x' > -2$ *and the formula* $C(X) \triangleq x \geq -10$. *The safety check for* $\langle Init, Tr, \neg C(X), X \rangle$ *returns* SAFE*, and the algorithm concludes that the transition system is terminating.*

Limitation of the approach. *Consider a different TS, with $X = \{x, u\}$, $Init(X) \triangleq u < 10 \wedge x > 0$, and $Tr(X, X') \triangleq ((x' = x-1 \wedge u' = u) \vee (x' = x+1 \wedge u' = u+1 \wedge u < 10)) \wedge x > 0$. Intuitively, each transition either decrements x (and keeps u unchanged), or increments both x and u until $u < 10$. Once $u = 10$, only the decrementing transition is possible, and x decreases monotonically to zero. Hence TS is terminating. The sink states are $Sink(X) \triangleq x \leq 0$.*

Algorithm 7 detects a trace $Tr^1(X^{(0)}, X^{(1)})$ reaching Sink states. TrInvGen *constructs transition invariant: $TrInv(X, X') \triangleq x' < x \wedge x > 0$. The coverage formula is computed as: $C(X) \triangleq x \leq 0 \vee (x > 0 \wedge u = 10)$. The safety check for $\langle Init, Tr, \neg C, X \rangle$ returns* UNSAFE*, since states with $u < 10$ are reachable from Init.* BlockTerm *then blocks the states that are guaranteed to reach sink: $Tr(X, X') \leftarrow Tr(X, X') \wedge x' > 0$.*

The algorithm calls SafetyCheck(P)*, which returns* SAFE *with inductive invariant $Inv(X) \triangleq x > 0$. However, Inv is not a recurrent set, so Algorithm 7 returns* Unknown.

The root cause why the algorithm cannot prove termination is that TrInv does not cover states with $u < 10$. It captures why the system terminates when $u = 10$, but not the global argument: u grows until at most $u = 10$, after which x decreases monotonically. A complete termination proof requires a second *disjunct covering the $u < 10$ region. Section 4.3 presents an extension that focuses on the state space for which the transition invariant candidate is not valid, aiming to extend the transition invariant.*

4.2.3 Correctness

This subsection proves the soundness of the introduced approach. It first proves that when the algorithm returns TERM, the transition system is terminating, and then proves the correctness for the nontermination.

Lemma 11. *If Algorithm 7 returns* TERM*, then the transition system $\langle Init, Tr, X \rangle$ is terminating.*

Proof. TERM is returned only at line 8. It means that there exists a transition invariant *TrInv* which is valid over states $C(X)$, and $\neg C(X)$ is unreachable from the initial states. The safety problem is updated iteratively, blocking the states that deterministically lead to termination, so in the end the $P = \langle Init', Tr', Sink, X \rangle$, where $Init'(X) = Init(X) \wedge \bigwedge_{i=0}^{m} \neg S_i(X)$ and $Tr' = Tr(X, X') \wedge \bigwedge_{j=0}^{l} \neg S_j(X')$. According to Lemma 10, transition system $\langle Init', Tr', X \rangle$ is terminating, since *TrInv* is a disjunctively well-founded transition invariant over all reachable states.

Since the states satisfying $\bigvee_{i=0}^{m} S_i(X)$ and $\bigvee_{j=0}^{l} S_j(X')$ deterministically lead to the termination, every trace of $\langle Init, Tr, X \rangle$ either follows Tr' or passes through a blocked state, which leads to termination. Hence $\langle Init, Tr, X \rangle$ is terminating. □

Lemma 12. *If Algorithm 7 returns* NONTERM*, then the transition system* $\langle Init, Tr, X \rangle$ *is nonterminating.*

Proof. The proof of this lemma follows directly from Lemma 9, as the nontermination analysis in Algorithm 7 follows exactly the same procedure as Algorithm 4. □

4.3 Extended Interpolation-based Transition Invariant Generation for Proving and Disproving Termination

Algorithm 7 is already capable of proving termination efficiently through the construction of transition invariants. This section describes an extension of the algorithm that aims to improve interaction between termination and nontermination analyses. Additionally, it presents further extensions to the original SNA algorithm that improve overall performance. The high-level flow of the extended algorithm can be seen in Figure 4.4.

4.3.1 Analysis of States for Which the Transition Invariant Is Not Valid

A transition invariant candidate can be used to guide the termination analysis. Particularly, Algorithm 7 constructs the formula $C(X)$ encoding states for which the transition invariant is valid. If $\neg C(X)$ is reachable, there exist states for which the transition invariant is not valid, and which can potentially lead to nontermination. The intuition is to compute the reachable states S_m within $\neg C(X)$ (which corresponds to line 11 of Algorithm 8). Then, it is possible to run a (non)termination check for these particular states within TS (line 12).

In the case of termination, the algorithm produces a witness transition invariant that extends the transition invariant of the original system, making it valid for more states. If nontermination is proved for S_m, then the original TS is also nonterminating, since S_m is reachable from the initial states.

4.3.2 Overall Algorithm

Algorithm 8 extends Algorithm 7 to enable more efficient (non)termination analysis. First, in addition to maintaining the transition invariant, the algorithm maintains a stack *Prs* of safety problems to be analyzed (line 2). At each iteration, the algorithm pops a safety problem $P' = \langle Init, Tr, Sink, X \rangle$ from the stack and analyzes it (line 4). The stack allows the algorithm to focus on (non)termination analysis of particular

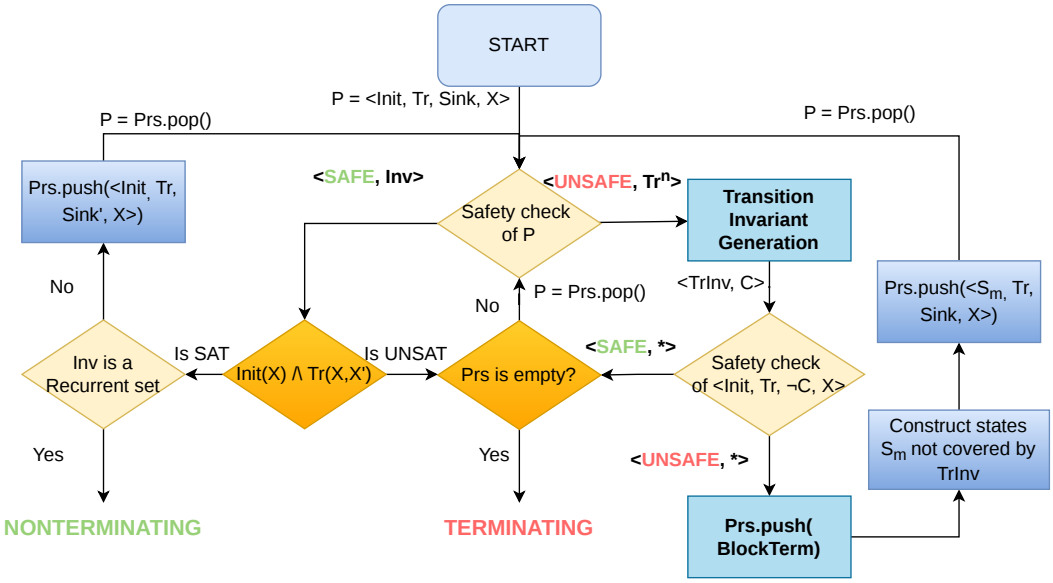


Figure 4.4. Extended Interpolation-based Termination Analysis (new components are highlighted in dark orange and dark blue colors).

reachable states within TS. The original safety problem always remains in the *Prs* stack.

The postprocessing phase of transition invariant construction is the main change to Algorithm 7. When $\neg C$ is reachable, the extended algorithm performs three actions: (1) it pushes the BlockTerm-refined problem (line 10); (2) it computes a formula encoding reachable states for which the transition invariant candidate is not valid (line 11); and (3) it pushes a new subproblem to analyze for termination (line 12), with S_m as the initial states. This allows the algorithm to concentrate on proving termination or nontermination of particular reachable states.

If $\neg C$ is not reachable, *TrInv* proves termination of the currently analyzed TS (the last TS popped from the stack), and the number of remaining subproblems decreases. If the stack is empty, Algorithm 8 returns TERM. The algorithm incorporates two further extensions.

Simple termination check (line 19). When the safety check returns SAFE and $\phi(X)$ is not a recurrent set, the algorithm additionally checks whether it is possible to take a transition from the initial states by querying:

$$Init(X) \wedge Tr(X, X')$$

If this formula is unsatisfiable, then no transition can be taken from the initial states (or all transitions are blocked, meaning all reachable states deterministically lead to termination), and the system is trivially terminating.

Algorithm 8: Extended Interpolation-based Transition Invariant Generation for Proving and Disproving Termination.

Input : Safety problem P
Output: TERM | NONTERM
Data : *TrInv* - candidate transition invariant, *Prs* - stack of safety problems

1 $TrInv \leftarrow \bot$
2 $Prs \leftarrow [P]$
3 **while** $\top$ **do**
4 $\quad P' \equiv \langle Init, Tr, Sink, X\rangle \leftarrow Prs.pop()$
5 $\quad \langle res, \phi\rangle \leftarrow$ SafetyCheck(P')
6 $\quad$ **if** $res =$ UNSAFE **then**
7 $\quad\quad \langle TrInv, C\rangle \leftarrow$ TrInvGen$(P', \phi, TrInv)$
8 $\quad\quad \langle res', \phi'\rangle \leftarrow$ SafetyCheck$(\langle Init, Tr, \neg C, X\rangle)$
9 $\quad\quad$ **if** $\langle res', \phi'\rangle = \langle$UNSAFE$, Tr^m\rangle$ **then**
10 $\quad\quad\quad Prs.push($BlockTerm$(P', \phi))$
11 $\quad\quad\quad S_m(X) \leftarrow QE(\exists X^{(0)}, \ldots, X^{(m-1)}.Init(X^{(0)}) \wedge \phi'(X^{(0)}, \ldots, X^{(m)}) \wedge \neg C(X^{(m)}))[X^{(m)} \mapsto X]$
12 $\quad\quad\quad Prs.push(\langle S_m, Tr, C, X\rangle)$
13 $\quad\quad$ **else**
14 $\quad\quad\quad$ **if** $Prs = \emptyset$ **then return** TERM
15 $\quad\quad\quad$ **else continue**
16 $\quad$ **else**
17 $\quad\quad$ **if** $\forall X.\phi(X) \rightarrow \exists X'.Tr(X, X') \wedge \phi(X')$ **then**
18 $\quad\quad\quad$ **return** NONTERM
19 $\quad\quad$ **if** UNSAT? $Init(X) \wedge Tr(X, X')$ **then**
20 $\quad\quad\quad$ **if** $Prs = \emptyset$ **then return** TERM
21 $\quad\quad\quad$ **else continue**
22 $\quad\quad Sink'(X) \leftarrow \neg QE(\exists X'.Tr(X, X'))$
23 $\quad\quad Prs.push(\langle Init, Tr, Sink', X\rangle)$

Refinement of Sink states (lines 22, 23). When $\phi(X)$ is not a recurrent set, this is often because BlockTerm has eliminated all transitions leading to *Sink*. This means that the produced ϕ contains sink states due to the updated *Tr*. The new formula encoding sink states is computed as:

$$Sink'(X) = \neg QE(\exists X'.Tr'(X, X'))$$

Formula $Sink'$ replaces *Sink* in the safety problem, enabling the analysis to proceed instead of returning UNKNOWN.

Lemma 13. *If Algorithm 8 returns* TERM*, then the original transition system* $\langle Init, Tr, X\rangle$ *is terminating.*

Proof. TERM is returned at line 14 or line 20. As in Algorithm 7, to return TERM, the termination conditions need to be satisfied for the transition system $\langle Init', Tr', X\rangle$, where $Init'(X) = Init(X) \wedge \bigwedge_i \neg S_i(X)$, and $Tr'(X,X') = Tr(X,X') \wedge \bigwedge_j \neg S_j(X')$. $\bigvee_{i=0}^{m} S_i(X)$ and $\bigvee_{j=0}^{l} S_j(X')$ are the states that deterministically lead to termination.

Case 1. If TERM is returned at line 14, then the termination proof follows directly from Lemma 11. This is the case, since the original safety problem is preserved in the stack, so if $Prs = \emptyset$, the original safety problem is terminating.

Case 2. If TERM is returned at line 20, then formula $Init'(X) \wedge Tr'(X,X')$ is unsatisfiable, so no transition is reachable from $Init'$. Thus, every transition in the original transition system leads to deterministically terminating states, and the transition system is terminating. □

Lemma 14. *If Algorithm 8 returns* NONTERM*, then the original transition system* $\langle Init, Tr, X\rangle$ *is nonterminating.*

Proof. NONTERM is returned at line 18, where $\phi(X)$ is a recurrent set for the currently considered TS $\langle Init', Tr', X\rangle$. By Lemma 9, there exists a recurrent set in the transition system $\langle Init', Tr', X\rangle$, and since $Tr'(X,X') \rightarrow Tr(X,X')$ and $Init'$ is reachable from $Init$ (or it is $Init$), this recurrent set witnesses nontermination of TS. □

4.4 Implementation and Evaluation

This section presents the implementation details and experimental evaluation of the approach.

4.4.1 Implementation

Prototypes of Algorithms 4, 7, and 8 were implemented inside Golem[2]; these implementations are referred to as SNA, ItpTig, and ItpTig+. Golem was chosen because it supports multiple different safety verification procedures, such as Property-Directed Reachability (PDR), Transition Power Abstraction (TPA), and Interpolation-based Model Checking. Additionally, the quantifier elimination already implemented in Golem is used. For SMT queries and interpolation, the implementation uses the OpenSMT solver. OpenSMT is integrated within Golem and provides native support for interpolation.

[2] https://github.com/usi-verification-and-security/golem.

Table 4.1. Comparison of ItpTig+ with SNA and ItpTig.

Algorithm	Total	Terminating	Nonterminating	Uniquely
SNA	343	0	343	7
ITPTIG	521	186	335	7
ITPTIG+	761	320	441	240

4.4.2 Evaluation

ITPTIG+ was compared against SNA and ITPTIG to evaluate the effect of combining termination and nontermination analyses. Additionally, ITPTIG+ was evaluated against several state-of-the-art tools: LOAT (version eacb74b), KOAT (version 41cb681), T2 (version 10f1373). The evaluation focused on termination analysis of Integer Transition Systems. Experiments were conducted on an Ubuntu 20.04 machine with an AMD EPYC 7452 32-core processor and 8x32 GiB RAM. The evaluation addressed the following research questions:

- **RQ1:** How does extending SNA with the termination procedure affect verification performance?
- **RQ2:** How does ITPTIG+ perform compared to state-of-the-art tools?

The benchmark suite of Integer Transition Systems was taken from the Termination Competition. Overall, there are 1222 benchmarks (538 nonterminating, 665 terminating, and 19 unknown). 42 trivially terminating benchmarks were excluded from the evaluation (termination of which can be detected by trivial syntactic checks), as they are solved by all tools. The evaluation was executed with a 120-second timeout (increasing the timeout does not significantly influence the evaluation).

Table 4.1 gives the experimental results for ITPTIG+, SNA, and ITPTIG. It confirms that ITPTIG+ consistently outperforms both SNA and ITPTIG for terminating and nonterminating instances, solving 240 instances that neither SNA nor ITPTIG could solve. This can be attributed to ITPTIG+'s generation of focused subproblems, which allows the analysis to concentrate on specific reachable states. This benefits recurrent set detection by focusing the analysis on specific states not covered by the transition invariant. It also benefits termination proving by extending the transition invariant with additional disjuncts that cover different regions of the reachable state space. In answer to **RQ1**, the combination of termination and nontermination reasoning in ITPTIG+ significantly outperforms either individual approach in isolation.

Interestingly, ITPTIG+ behaves differently depending on the underlying safety verification engine: with TPA it solved 761 instances and with PDR – 755. That said, it solved 24 (resp. 18) benchmarks uniquely with TPA (resp. PDR).

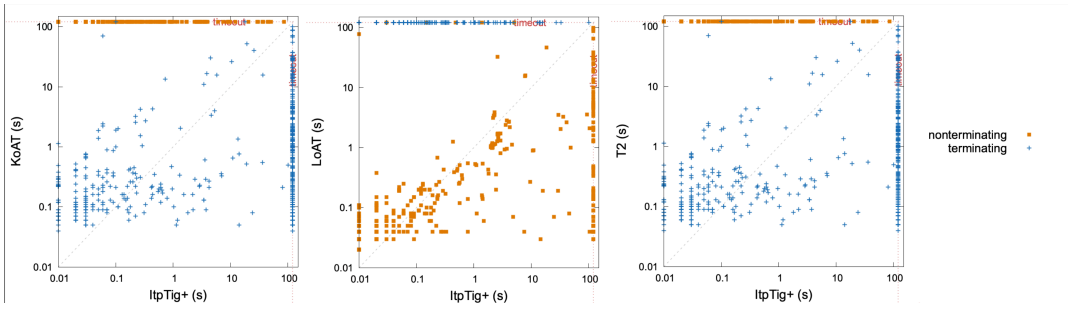


Figure 4.5. Comparison of ItpTig+ with KoAT, LoAT and T2 (runtime in seconds).

Table 4.2. Comparison of ItpTig+ with state-of-the-art tools.

Algorithm	Total	Terminating	Nonterminating	Uniquely
LoAT	525	0	525	48
KoAT	562	562	0	26
T2	1002	572	430	40
ItpTig+	761	320	441	8

Figure 4.5 shows a scatter plot comparing ItpTig+ against state-of-the-art termination analyzers, where each axis represents the solving time in seconds for a given tool. Orange squares denote nonterminating instances; blue crosses denote terminating instances. Points above the diagonal indicate instances where ItpTig+ is faster than the competitor. ItpTig+ solves 761 benchmarks, with T2 solving 1002, KoAT 562, and LoAT 525 instances; the full comparison can be seen in Table 4.2. As shown in Figure 4.5, ItpTig+, despite being a prototype implementation, is competitive with state-of-the-art termination analyzers. Among all compared tools, ItpTig+ uniquely solves eight instances. Notably, two of these — `juLinkedListCreateAddAll.jar-obl-11` and `juLinkedListCreateAddAllAt.jar-obl-17` — have never been solved in the Termination Competition. ItpTig+ found recurrent sets and proved nontermination for them. In answer to **RQ2**, ItpTig+ is competitive with state-of-the-art tools and solves instances not solved by any tool in the Termination Competition.

4.5 Limitations

The evaluation also revealed limitations of the approach.

Complex transition relations. The transition invariant is constructed by generalizing individual terminating traces with Craig interpolation. When the transition relation is complex, e.g., when it consists of many disjuncts corresponding to different

branches of a loop body, an interpolant computed for one trace may describe only the branches taken by that trace. The algorithm then needs many iterations, each requiring a new safety check, before the transition invariant covers all reachable behaviors. Moreover, the interpolant is converted into DNF before the well-foundedness of its disjuncts is checked, and the number of disjuncts can grow exponentially in the size of the interpolant. For example, for a loop whose body consists of k consecutive `if-else` statements, each updating a different variable, the interpolants may need to distinguish up to 2^k combinations of branches.

Quantifier elimination. The approach makes heavy use of quantifier elimination: to compute the sink states, the sets of states S_i along a terminating trace in BlockTerm, and the region $C(X)$ where the transition invariant is valid. Each of these eliminates the variables of an entire trace, so its cost grows with the length of the trace, and quantifier elimination for linear integer arithmetic is expensive in the worst case. Since TPA is designed to find long traces, a single terminating trace of, say, a thousand steps leads to a quantifier elimination problem over a thousand copies of the program variables. In the experiments, such quantifier elimination problems were a frequent cause of timeouts.

Possible remedies, such as a modular analysis of individual loops and cheaper alternatives to quantifier elimination, are discussed in Section 4.7.

4.6 Related Work

Recent years have seen significant advances in automated techniques for both termination and nontermination analysis. The concrete citations of the related work are introduced in the Chapter 2 and above in this chapter.

Nontermination. Nontermination analysis is typically centered on the construction of recurrent sets — sets of states from which execution cannot escape. These techniques range from template-based synthesis to safety-check-based recurrent set construction, the latter of which the approach in this chapter directly extends. An alternative approach detects nontermination by identifying lasso-shaped executions — finite paths leading to a cycle — via bounded model checking. Leike and Heizmann use geometric nontermination witnesses: finite representations of infinite executions via sums of geometric series. Unlike the approaches above, which treat nontermination in isolation, the technique in this chapter is integrated with termination analysis within a unified framework, so that intermediate results guide one another.

Termination. The most widely studied approach to termination analysis is the synthesis of ranking functions. State-of-the-art techniques focus on the construction of multiphase or lexicographic ranking functions, which significantly broaden the class of

programs for which termination can be proved. However, the complexity of a transition system can make ranking function synthesis infeasible. Fixpoint-based methods reduce termination and nontermination to the validity of a first-order fixpoint formula. Disjunctively well-founded transition invariants enable overapproximating the behavior of the transition system. The central challenge there is the generation of the transition invariant itself: existing techniques rely either on template-based generation or on the direct construction of a ranking relation from a particular trace. In contrast, the approach in this chapter uses Craig interpolation to guide transition invariant synthesis with respect to the actual termination conditions, producing property-directed invariants.

4.7 Future Work

One way to extend this work is to employ the modular techniques described in Chapter 3. A modular approach would significantly improve the performance of termination analysis. One of the core weaknesses of the technique is the cost of interpolant production and the search for invariants during the safety check. A more modular approach would allow every loop in the system to be analyzed in isolation, simplifying both the interpolation and the invariant, since the analyzed system would be much simpler.

The cost of quantifier elimination (Section 4.5) could be reduced by replacing exact quantifier elimination with model-based projection where an underapproximation suffices, e.g., when blocking terminating states, and by working with shorter trace segments instead of complete traces.

Another possible direction is to use different ranking arguments for the well-foundedness check of the disjuncts. Currently, the well-foundedness test relies on a linear ranking function. This function could be replaced with lexicographic, multiphase, or non-linear ranking arguments, as the algorithm does not depend on any specific ranking-function synthesis approach. Richer ranking arguments would let more disjuncts pass the well-foundedness check, admitting more candidates into the transition invariant.

4.8 Conclusion

This chapter has presented an interpolation-based approach to transition invariant generation for proving and disproving termination of infinite-state systems. The approach extends safety-based nontermination analysis with a termination proving capability, unifying the two within a single framework. The key insight is to exploit terminating traces produced by the SNA as a source of structural information, using Craig interpolation to construct well-founded transition invariants guided by the termination con-

ditions. The interaction between termination and nontermination reasoning is central to the technique: transition invariants focus the nontermination analysis on uncovered states, while SNA provides terminating traces that drive invariant generation. Grounding the approach in safety verification provides a robust foundation, enabling efficient checking of both termination and nontermination via existing safety solvers. The soundness of the algorithm was proved, and an extensive experimental evaluation was conducted. The results demonstrate the benefit of combining termination and nontermination analysis, and show that the approach achieves competitive performance against state-of-the-art tools, including the unique solution of instances never solved by any tool in the Termination Competition. Using this approach, Golem received an award for the tool with the biggest improvement in the Termination Competition 2026 [98].

Chapter 5

Safety-based Automated Test Generation

Safety verification can have many different applications other than proving concrete properties of a program. Sometimes complete safety verification can be infeasible or even impossible in the case of a complex software system. In these cases, other techniques, such as testing, can be used. Safety-verification-based counterexample search can be very effective for the automated test generation. This chapter presents an approach for safety-based automated test generation for Solidity programs. Background on smart contracts and on existing approaches to their analysis is given in Section 2.3.

As discussed in Chapter 2, CHCs provide an efficient logic-based format for verification. The CHC-based solutions focus on the one hand on the development of a front-end for the translation of the source code into the language of logic constraints and on the other hand on the implementation of the back-end for solving logical queries constructed from the encoding. CHC solvers have been applied to verification of programs in various languages (e.g., SeaHorn, Korn and TriCera for verification of C code, JayHorn for Java, RustHorn for Rust [141], SolCMC and SmartACE [177] for Solidity). Recently, the CHC-based approach called Horntinuum was also proposed for the automated exhaustive test-case generation for an imperative language without recursion [182]. Given a CHC-encoding, this approach gradually explores various control-flow paths represented by CHC unrollings and relies on an off-the-shelf SMT solver to produce exact values for the program inputs.

This chapter builds on top of [182], proposing a framework for test generation for programs in Solidity, a contract-oriented language, and on an implementation of an automated test generation tool on top of the Solidity compiler's model checker, SolCMC. It relies on a CHC representation of a smart contract generated by SolCMC which is different from the encoding of an imperative program, due to the presence of functions that can be invoked in any order. This complicates the path enumeration process, necessary for generating test cases. The new approach explores contract be-

haviors gradually, from shorter ones to longer ones, and keeps track of functions and branches that have already been covered.

This chapter presents the algorithmic solution as it was implemented in the SolTG, a new fully automated tool for Solidity test generation. SolTG receives one or more Solidity source files, extracts the compiler metadata and the CHC representation from SolCMC, computes CHC unrollings, and communicates with the Z3 SMT solver to receive values of function parameters for a test case. Each test case is compiled into a human-readable test file following the format of the well-known Foundry[1] framework to build, test, fuzz, debug, and deploy Solidity smart contracts. Thus, it becomes immediately useful for contract engineers, who receive test coverage reports based on the contract execution that cannot be obtained with other tools. This contribution was published and presented at CAV 2024 [50].

5.1 SolTG: Test Case Generator for Solidity

5.1.1 Tool Overview

Figure 5.1 gives an overview of the architecture of SolTG[2]. The input is a source file with a Solidity smart contract, and the output is a set of test files for Foundry, a framework to build, test, fuzz, debug, and deploy Solidity smart contracts. The tool relies on the external modules: **Solidity Compiler** to obtain the compilation metadata, **Solidity Compiler Model Checker** to get the CHC encoding, and an SMT solver **Z3** to actually extract concrete values for test cases.

Figure 5.2 gives an overview of a higher-level testing process monitored by engineers, where SolTG plays the central role. The generated test files are human readable and are kept and used for testing until the contract under test has been updated. The test reports in the HTML format (Figure 5.3) are generated by Foundry, which compiles the contracts being tested and executes the test cases. Reports provide information on three different types of coverage for each specific benchmark. Additionally, the average coverage metrics for the whole project are measured (seen in the top right corner). The internal components of SolTG are described below.

Preprocessor is responsible for parsing and analyzing the input data. It receives compilation metadata and the CHC encoding of the Solidity contract. Preprocessor then parses the compiler's metadata, determining the complete list of the contract's

[1] https://book.getfoundry.sh.

[2] The code is available at https://github.com/usi-verification-and-security/SolTG (the backend is available at https://github.com/BritikovKI/aeval/tree/tg-nonlin); the artefact is available at https://zenodo.org/records/17061325.

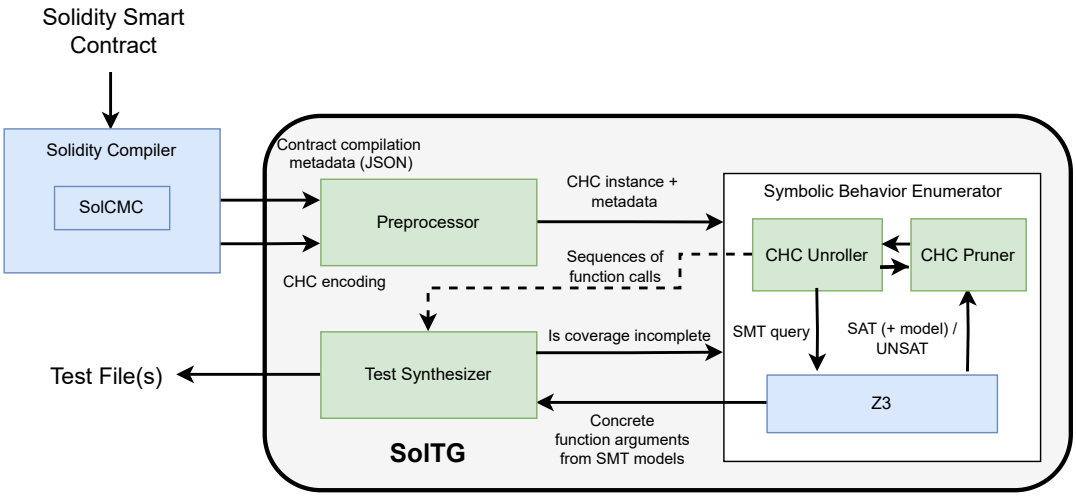


Figure 5.1. Architecture of SolTG.

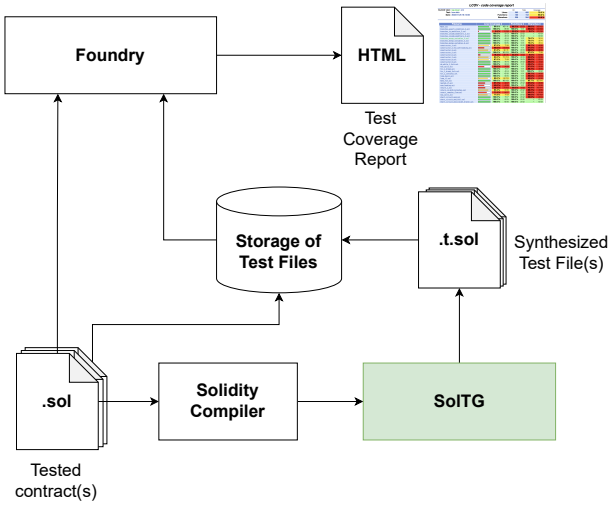
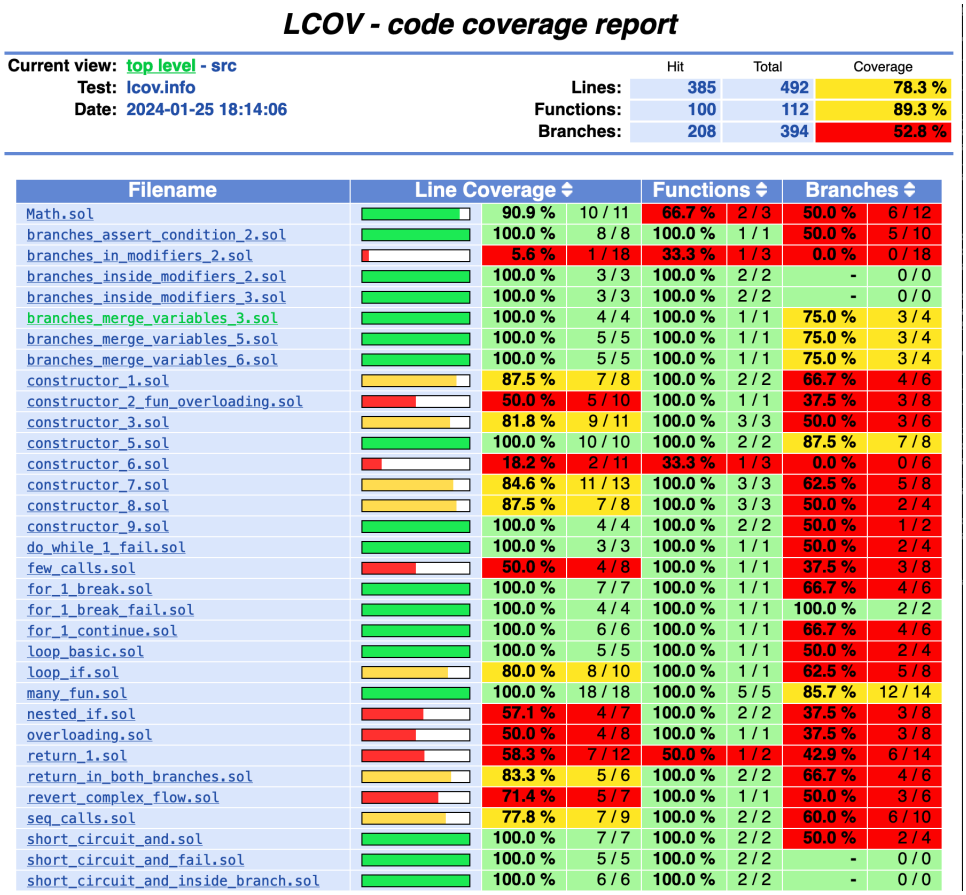


Figure 5.2. High-level testing workflow with SolTG.

public and external (testable) functions, constructors, and their required inputs. Preprocessor encodes the full set of constructors, functions, and input variables, and then provides it as input together with the CHC encoding to Symbolic Behavior Enumerator to discover possible execution scenarios.

Symbolic Behavior Enumerator (see details in Sect. 5.2) synthesizes sequences of function calls over tuples of concrete values for function parameters. Overall, SolTG exhaustively enumerates symbolic representations of different paths through the contract's functions and relies on an SMT solver (here, Z3) to extract values of those pa-



LCOV - code coverage report

		Hit	Total	Coverage
Current view: top level - src	Lines:	385	492	78.3 %
Test: lcov.info	Functions:	100	112	89.3 %
Date: 2024-01-25 18:14:06	Branches:	208	394	52.8 %

Filename	Line Coverage		Functions		Branches	
Math.sol	90.9 %	10 / 11	66.7 %	2 / 3	50.0 %	6 / 12
branches_assert_condition_2.sol	100.0 %	8 / 8	100.0 %	1 / 1	50.0 %	5 / 10
branches_in_modifiers_2.sol	5.6 %	1 / 18	33.3 %	1 / 3	0.0 %	0 / 18
branches_inside_modifiers_2.sol	100.0 %	3 / 3	100.0 %	2 / 2	-	0 / 0
branches_inside_modifiers_3.sol	100.0 %	3 / 3	100.0 %	2 / 2	-	0 / 0
branches_merge_variables_3.sol	100.0 %	4 / 4	100.0 %	1 / 1	75.0 %	3 / 4
branches_merge_variables_5.sol	100.0 %	5 / 5	100.0 %	1 / 1	75.0 %	3 / 4
branches_merge_variables_6.sol	100.0 %	5 / 5	100.0 %	1 / 1	75.0 %	3 / 4
constructor_1.sol	87.5 %	7 / 8	100.0 %	2 / 2	66.7 %	4 / 6
constructor_2_fun_overloading.sol	50.0 %	5 / 10	100.0 %	1 / 1	37.5 %	3 / 8
constructor_3.sol	81.8 %	9 / 11	100.0 %	3 / 3	50.0 %	3 / 6
constructor_5.sol	100.0 %	10 / 10	100.0 %	2 / 2	87.5 %	7 / 8
constructor_6.sol	18.2 %	2 / 11	33.3 %	1 / 3	0.0 %	0 / 6
constructor_7.sol	84.6 %	11 / 13	100.0 %	3 / 3	62.5 %	5 / 8
constructor_8.sol	87.5 %	7 / 8	100.0 %	3 / 3	50.0 %	2 / 4
constructor_9.sol	100.0 %	4 / 4	100.0 %	2 / 2	50.0 %	1 / 2
do_while_1_fail.sol	100.0 %	3 / 3	100.0 %	1 / 1	50.0 %	2 / 4
few_calls.sol	50.0 %	4 / 8	100.0 %	1 / 1	37.5 %	3 / 8
for_1_break.sol	100.0 %	7 / 7	100.0 %	1 / 1	66.7 %	4 / 6
for_1_break_fail.sol	100.0 %	4 / 4	100.0 %	1 / 1	100.0 %	2 / 2
for_1_continue.sol	100.0 %	6 / 6	100.0 %	1 / 1	66.7 %	4 / 6
loop_basic.sol	100.0 %	5 / 5	100.0 %	1 / 1	50.0 %	2 / 4
loop_if.sol	80.0 %	8 / 10	100.0 %	1 / 1	62.5 %	5 / 8
many_fun.sol	100.0 %	18 / 18	100.0 %	5 / 5	85.7 %	12 / 14
nested_if.sol	57.1 %	4 / 7	100.0 %	2 / 2	37.5 %	3 / 8
overloading.sol	50.0 %	4 / 8	100.0 %	1 / 1	37.5 %	3 / 8
return_1.sol	58.3 %	7 / 12	50.0 %	1 / 2	42.9 %	6 / 14
return_in_both_branches.sol	83.3 %	5 / 6	100.0 %	2 / 2	66.7 %	4 / 6
revert_complex_flow.sol	71.4 %	5 / 7	100.0 %	1 / 1	50.0 %	3 / 6
seq_calls.sol	77.8 %	7 / 9	100.0 %	2 / 2	60.0 %	6 / 10
short_circuit_and.sol	100.0 %	7 / 7	100.0 %	2 / 2	50.0 %	2 / 4
short_circuit_and_fail.sol	100.0 %	5 / 5	100.0 %	2 / 2	-	0 / 0
short_circuit_and_inside_branch.sol	100.0 %	6 / 6	100.0 %	2 / 2	-	0 / 0

Figure 5.3. HTML coverage report.

rameters, until either all branches are covered with tests or a timeout is reached.

Test Synthesizer receives a set of concrete function arguments for the tests from the SMT models and corresponding sequences of function calls that are provided by Symbolic Behavior Enumerator. It processes the inputs and generates a set of tests stored as a test file for FOUNDRY.

FOUNDRY compiles, executes, performs coverage analysis, and generates a report for the synthesized test files. Coverage analysis uses classical test coverage metrics, such as line/branch/function coverage, which are commonly used by contract engineers. The distinguishing feature of the framework is that the reported results reflect the exact blockchain response to the tested function calls.

5.1.2 Support of Solidity Constructs

SolTG supports various Solidity features: it can process smart contracts with constructors, multiple fields and functions, polymorphism, inheritance of other contracts or interfaces, etc. It can also handle Solidity-specific constructs like inherent transaction data, the current state of the blockchain, contract state variables, and a full set of standard Solidity datatypes. A set of features unique to the Ethereum blockchain and the Solidity programming language is outlined below.

Addresses Addresses are one of the most used data structures in the development of smart contracts. An address is a 20-byte array that represents a user or a smart contract in the blockchain space, and it is encoded as an integer number. Resulting tests with addresses have the following form:

```
1 function test_weth() public {
2     weth.approve(0x52F6000000000000000000000000000000000000, 7719)
      ;
3     assertTrue(true);
4 }
```

Transaction data Every function call in Solidity is a complex data structure. Every function call has a sender, receiver (which are the addresses in the blockchain space), value, data, and many other parameters. SolTG is capable of modeling those intrinsic function call parameters if they influence function execution. For example, for the function:

```
1     function deposit() public payable {
2         if(msg.value > 1e18){
3             balanceOf[msg.sender] += msg.value;
4         }
5     }
```

SolTG would generate 2 tests, which fully cover branching on *msg.value*:

```
1     function test_weth_1() public {
2         weth1.deposit();
3         assertTrue(true);
4     }
5     function test_weth_2() public {
6         weth2.deposit{ value: 1000000000000000001 wei }();
7         assertTrue(true);
8     }
```

Blockchain state support Analogous to the message data, the blockchain also has its current state at the point of transaction execution. Specifically, this includes information about the current block, timestamp, the balances of all addresses, address data (specific to the smart contracts), and other parameters. Due to SolCMC-provided encoding, SolTG currently supports only the balances of the addresses. Consider the following example:

```
1     function balanceCheck()
2         public
3         returns (bool)
4     {
5         if(address(this).balance > 1000){
6             return true;
7         } else {
8             return false;
9         }
10    }
```

SolTG would be able to generate 2 different tests for this specific branching, as the balance of the specific address is also extracted from the model:

```
1   function test_store_1() public {
2         vm.prank(address(storage1));
3         vm.deal(address(storage1), 8720 wei);
4         storage1.balanceCheck();
5         //balanceCheck__33("address(this).balance=8720", 0)
6         assertTrue(true);
7   }
```

The test is generated automatically, and lines 2 and 3 serve specifically to mock the blockchain state and to assign a specific value to the ether balance associated with the address.

Contract state variables support Contracts frequently have internal variables on which the function execution in the contract might depend. It is possible to extract these variables from a model and assign them before the function call, so the function call has the behavior that corresponds to the one discovered by the Symbolic Behavior Enumerator.

```
1     uint UNIQUE_STATE_VAR;
2
3     function balanceCheck() public view returns (bool) {
4         if(UNIQUE_STATE_VAR > 1000){
5             assert(UNIQUE_STATE_VAR > 1000);
6             return true;
7         } else {
8             return false;
```

```
9           }
10      }
11
12      function set_state_var(uint n) public payable {
13          UNIQUE_STATE_VAR = n;
14      }
```

SolTG detects a state variable *UNIQUE_STATE_VAR* while processing the Solidity file, which leads to the generation of chains of functions that allow full coverage of *balanceCheck*. The tests look as follows:

```
1   function test_store_state_1() public {
2     storage1.balanceCheck();
3     assertTrue(true);
4   }
5   function test_store_state_2() public {
6     storage2.set_state_var( 1001);
7     storage2.balanceCheck();
8     assertTrue(true);
9   }
```

By calling *set_state_var* in the test, SolTG unblocks one of the branches in *balanceCheck* by setting a specific value for *UNIQUE_STATE_VAR*.

5.2 Test Case Generation from CHC Encoding

5.2.1 Solidity Smart Contracts to CHCs

This section begins with a brief intuitive overview of the CHC encoding employed by SolTG, which is inspired by SolCMC. For demonstration purposes, the encoding principles are simplified, but the main ingredients are highlighted. Each test-case generation target may operate over Solidity's ABI and cryptographic functions, transaction data, the blockchain state, balances, and storage for every contract. Their logic representation is denoted $\vec{s}$, and there is assumed to be a fully interpreted formula $init(\vec{s})$ that describes how $\vec{s}$ is instantiated *before* an instance of the contract under test has been created. The contract under test is assumed to have a vector of fields $v_1, \ldots, v_n$ that is symbolically encoded into some $\vec{v}$ that does not overlap with $\vec{s}$. Throughout the lifecycle of an instance of the contract, it could undergo a (possibly unbounded) number of changes made by calling the contract's functions. These are modeled with the help of an auxiliary uninterpreted predicate *fc*. Intuitively, it corresponds to a *contract transition system* (this notation is used later) that has two logic rules, for the initiation and consecution, respectively:

$$init(\vec{s}) \implies fc(\vec{s}, \vec{v}) \tag{5.1}$$

$$fc(\vec{s},\vec{v}) \wedge sum(\vec{s},\vec{s'},\vec{v},\vec{v'}) \implies fc(\vec{s'},\vec{v'}) \tag{5.2}$$

For the second rule, assume a set of defined functions F. An uninterpreted predicate *sum* stands for a *summary* of an arbitrary function over two vectors of logic variables $\vec{v}$ and $\vec{v'}$ that represent symbolic values of the contract's fields before and after each function call, respectively.

Each $f \in F$ has a CFG (V_f, E_f), where V_f is a set of basic blocks and E_f is a set of control-flow edges, each one connecting two basic blocks. The dedicated basic block $\texttt{en}_f$ is the only one with no incoming edge in E_f, and the dedicated basic block $\texttt{ret}_f$ is the only one with no outgoing edge in E_f. There is also assumed to be an encoding function $\tau : E_f \to Prop$ that translates each $(\texttt{b}_1, \texttt{b}_2) \in E_f$ into an SSA form and further to a logic formula over $\vec{s},\vec{s'},\vec{v},\vec{v'},\vec{in},\vec{loc}$, where $\vec{in}$ is a vector of input variables to f and $\vec{loc}$ is a vector of fresh local variables (and auxiliary SSA variables created during the encoding). This formula symbolically encodes the basic block $\texttt{b}_1$ and the condition under which the control transitions to basic block $\texttt{b}_2$. The CHC encoding then has a set of rules for f:

For each control-flow edge $(\texttt{b}, \texttt{ret}_f)$, the CHC encoding uses an uninterpreted predicate b:

$$\tau(b, ret_f) \implies b(\vec{s},\vec{s'},\vec{v},\vec{v'},\vec{in},\vec{loc}) \tag{5.3}$$

Then for each control-flow edge $(\texttt{b}_1, \texttt{b}_2)$ where $\texttt{b}_2 \neq \texttt{ret}_f$, the CHC encoding uses uninterpreted predicates b_1 and b_2:

$$b_2(\vec{s},\vec{s'},\vec{v},\vec{v'},\vec{in},\vec{loc}) \wedge \tau(b_1, b_2) \implies b_1(\vec{s},\vec{s'},\vec{v},\vec{v'},\vec{in},\vec{loc}) \tag{5.4}$$

Lastly, to connect the entry to function f with its summary, it uses uninterpreted predicates en_f and *sum* (which is in turn used to connect f with the contract transition system via rule (5.2)):

$$en_f(\vec{s},\vec{s'},\vec{v},\vec{v'},\vec{in},\vec{loc}) \implies sum(\vec{s},\vec{s'},\vec{v},\vec{v'}) \tag{5.5}$$

This encoding is illustrated using the following example contract with one field `a`, a constructor, and two other functions.

```
1 contract A{
2   uint a;
3   constructor(uint in) {a = in;}
4   function reset() public {a = 0;}
5   function f(uint x, uint y) public {
6     if (a < x) reset();
7     else a += y;}
8 }
```

The CHC encoding is as follows. Field `a` (resp. inputs `in`, `x`, and `y`) is represented in CHCs by logic variable a (resp. in, x, and y). For simplicity, $\vec{s}$ is taken to be empty and *init* to be *true*. The entry points to the constructor and two functions are represented using predicates en_{con}, en_{reset}, and en_f, respectively. The CFG of the constructor (similarly, function `reset`) is trivial, so it needs one CHC of form (5.3) and one of form (5.5). Since function `f` contains a conditional statement, its CFG has basic blocks $\mathtt{b}_1$ and $\mathtt{b}_2$, and its CHC encoding makes use of predicates b_1 and b_2 in the CHCs of form (5.3) and (5.4). For readability, formula $\tau(\mathtt{en}_f, \mathtt{b}_1)$ (resp. $\tau(\mathtt{en}_f, \mathtt{b}_2)$) is colored purple (resp. blue) and placed in the box. Lastly, note that $\tau(\mathtt{b}_1, \mathtt{ret}_f)$ has an occurrence of predicate en_{reset}, which corresponds to calling function `reset`.

$$
\begin{aligned}
true &\implies fc(a) \\
fc(a) \wedge sum(a,a') &\implies fc(a') \\
\boxed{a' = in} &\implies en_{con}(a,a',in) \\
en_{con}(a,a',in) &\implies sum(a,a') \\
\boxed{a' = 0} &\implies en_{reset}(a,a') \\
en_{reset}(a,a') &\implies sum(a,a')
\end{aligned}
\qquad
\begin{aligned}
en_{reset}(a,a') &\implies b_1(a,a',x,y) \\
\boxed{a' = a + y} &\implies b_2(a,a',x,y) \\
b_1(a,a',x,y) \wedge \boxed{a < x} &\implies en_f(a,a',x,y) \\
b_2(a,a',x,y) \wedge \boxed{a \geq x} &\implies en_f(a,a',x,y) \\
en_f(a,a',x,y) &\implies sum(a,a')
\end{aligned}
$$

The boxed formulae are the building blocks to be used in the test-code generation as described in the next subsection. Purple (resp. blue) color emphasizes that the formula is used to symbolically encode the `then`-branch (resp. the `else`-branch) of `f`'s behavior, and black color in the box indicates the code of the constructor.

5.2.2 Algorithmic Enumeration of Contract Behaviors

SolTG generates tests by symbolic enumeration of the possible behaviors of the contract under test. Each test case begins with creating an instance of the contract `c` (by calling its constructor). Suppose then we wish to observe the behavior of a certain function `f` which will be expressed as `c.f(…)` in the test. If `f` uses an input, the test case should specify one of the possible concrete values. A test could have a sequence of multiple functions under certain inputs, each of which contributes to updating the fields. For the example above, when we wish to test function `f`'s behavior, we could (manually) create a test `c = new A(0); c.f(0,0)`. This would, however, not be enough to test the `then`-branch of the conditional in `f`'s body. For this reason, another test, e.g., `c = new A(0); c.f(1,0)` would be needed. In general, finding these sequences of function calls and concrete inputs is challenging. Our tool effectively generates sequences of function calls by enumeration over permutations of available functions, and it generates concrete inputs from the models of satisfiable logic formulae that correspond to distinct unrollings of CHCs. Specifically, the enumeration has three

nested loops (denoted respectively **A**, **B**, and **C**).

***A:** Enumerating the lengths of tests* At the higher level, the tool sequentially considers *unfoldings* of the contract transition system (5.1)–(5.2), which allows it to eventually consider test cases of various lengths. That is, an unrolling of length one would yield only test cases with constructors, an unrolling of length two – test cases with a constructor and one function call, etc. More formally, at this level, the tool performs an unfolding of fc in CHC $fc(v) \wedge sum(v, v') \implies fc(v')$ but keeps sum uninterpreted. For a length n, the tool unfolds fc exactly $n+1$ times, for which it uses the consecution rule n times and then the initiation rule once. As a result, we get a new CHC (called C_n) with n distinct occurrences of sum in the body.

***B:** Enumerating the functions* At the middle level, i.e., for an unrolling of a fixed length n, the tool considers a sequence of n functions (possibly with repetitions), where the first function is necessarily a constructor. Specifically, given the output from the outer loop, i.e., a CHC C_n with n distinct occurrences of sum in the body, the tool considers a set of n-tuples F^n and computes $|F|^n$ tuples of uninterpreted predicates $T = \{en_f \mid f \in F\}^n$. It finally computes $|F|^n$ distinct CHCs, each one by the pairwise replacement of the n-tuple of sum predicates in C_n by an n-tuple from T (introducing fresh variables for in and loc whenever needed).

***C:** Enumerating behaviors for each function* At the inner level, for each of the n functions to be tested, the tool considers all paths through the CFG. That is, it receives one of the CHCs constructed at the middle level, and it computes all possible *unrollings* by recursively eliminating all uninterpreted symbols. Algorithmically, an unrolling of each function is similar to the approach of, and the tool repeats it n times, for each high-level function call (i.e., each one synthesized in the test file). This loop gives a set of logic formulae, each of which is sent to an SMT solver. Lastly, to enumerate inputs for each function call, the tool relies on an SMT solver. Given a formula, the solver aims to find a satisfying assignment from which the desired input values are extracted.

Pruning test case enumeration Because of the exhaustiveness of enumeration, the complexity of steps **A**, **B**, and **C** grows exponentially with the number of contract functions and the points of control-flow divergence. The tool attempts to mitigate this by employing optimizations, similar to those in, to prune the search space of test cases. First, the initial function to be called in a test case should always be the constructor of the contract under test. Second, assume there is a subset $F' \subseteq F$ of functions that already have full coverage (thanks to the already synthesized test cases). Then, when synthesizing a new test case as a sequence of n function calls, step **B** can enumerate only $|F|^{n-2} \cdot |F \setminus F'|$ possibilities instead of $|F|^n$ (i.e., any function $f \in F'$ could only be called in either the 1-st, 2-nd, ..., or $n-1$-th position).

Synthesizing tests Each test case is synthesized from two components: a sequence of functions created at the end of step **B**, and a sequence of tuples of concrete values

extracted from an SMT model at the end of step **C**. In general, the tool has to create an unrolling of the following CHC, for some $f_1, f_2, \ldots \in F$:

$$en_{f_1}(\vec{s}, \vec{s'}, \vec{v}, \vec{v'}, \vec{loc}, \vec{in}) \wedge en_{f_2}(\vec{s'}, \vec{s''}, \vec{v'}, \vec{v''}, \vec{loc'}, \vec{in'}) \wedge \ldots \implies fc(\vec{s^{(n)}}, \vec{v^{(n)}})$$

Note that the ultimate unrolling still has occurrences of variables $\vec{in}$ and $\vec{in'}$ but no predicates en_{f_1}, $en_{f_2}, \ldots$ Thus, if satisfiable, the SMT solver returns a tuple of values $\texttt{in}_1, \texttt{in}_2, \ldots$ for $\vec{in}$, a tuple of values $\texttt{in'}_1, \texttt{in'}_2, \ldots$ for $\vec{in'}$, etc. The final test case is then constructed as follows:

$$\texttt{f}_1(\texttt{in}_1, \texttt{in}_2, \ldots); \texttt{f}_2(\texttt{in'}_1, \texttt{in'}_2, \ldots); \ldots$$

The whole process is illustrated on the running example. The formula encoding calls to the constructor and function `f` under the `then`-branch has the following unrolling:

$$\begin{aligned} a' = in \wedge \quad & \text{(}\texttt{constructor}\text{ – first call)} \\ a' < x \wedge \quad & \text{(}\texttt{f(x, y)}\text{ – second call, }\texttt{then}\text{-branch)} \\ a'' = 0 \quad & \text{(}\texttt{reset()}\text{ – nested call)} \end{aligned}$$

It is satisfiable with a model $in \mapsto 0, x \mapsto 1, y \mapsto 0 \ldots$ Parsing this model and determining that in represents an input to the constructor and x represents an input to `f` yields the test case `c = new A(0); c.f(1,0)`.

A formula encoding the same function calls but another branch is as follows:

$$\begin{aligned} a' = in \wedge \quad & \text{(}\texttt{constructor}\text{ – first call)} \\ a' \geq x \wedge a'' = a' + y \quad & \text{(}\texttt{f(x, y)}\text{ – second call, }\texttt{else}\text{-branch)} \end{aligned}$$

It is satisfiable with a model $in \mapsto 0, x \mapsto 0, y \mapsto 0 \ldots$, which yields the test case `c = new A(0); c.f(0,0)`. These test cases now can be used for the generation of a Solidity test file:

```
1 contract A_Test is Test {
2   A a0; A a1;
3   function setUp() public { a0 = new A(0); a1 = new A(0); }
4   function test_A_0() public { a0.f(1, 0); }
5   function test_A_1() public { a1.f(0, 0); } }
```

The test file consists of multiple functions, the naming convention for which follows the FOUNDRY[3] standards. `setUp` is used by FOUNDRY to prepare a testing environment. Functions `test_A_0` and `test_A_1` incorporate test cases generated by SOLTG above. The test file is human-readable and reusable: it can be used to generate test coverage right away and/or stored for regression testing later.

[3] https://book.getfoundry.sh/forge/writing-tests.

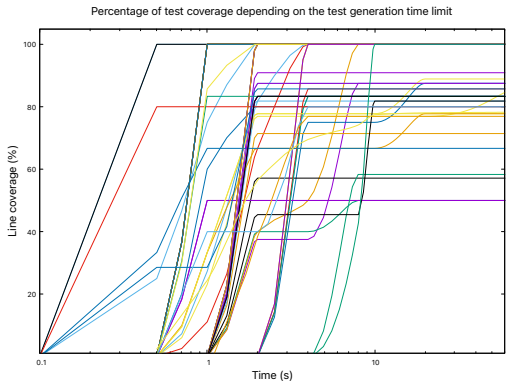


Figure 5.4. SolTG performance for the benchmark set with different timeouts.

5.3 Evaluation

SolTG was evaluated on 59 benchmarks from the SolCMC repository and industrial smart contracts that exhibit different Solidity-specific features. All experiments were conducted on a machine with a 2.3GHz 8-core Intel Core i9 processor and 16 GiB of memory, running Mac OS. Test coverage data was collected from the Foundry test reports.

For the given set of benchmarks, SolTG provided a high level of test coverage. In particular, on average, it achieved 81.2% line coverage and 90.9% function coverage for a 60-second timeout. A correlation of the coverage and the chosen timeout for each benchmark is shown in Figure 5.4. The cactus plot has a range of timeouts from 0.1 sec to 60 sec on the x-axis and the corresponding coverage on the y-axis. Each curve represents multiple behaviors of SolTG on a single benchmark w.r.t. different timeouts. The rapid growth of many curves demonstrates that (1) the preprocessing and initial enumeration steps are very fast, and (2) SolTG often needs just a few seconds to produce many test cases. In fact, SolTG generates the majority of tests within the first 10 seconds of the execution. For 21 benchmarks, SolTG generated tests that report 100% line coverage within the first 5 seconds. Further, as more time is allowed, SolTG finds additional test cases, but their production expectedly becomes more costly.

The cases where SolTG did not report full test coverage within the 60-second timeout were investigated. One reason, as expected, was the complexity of the control structures of the contracts, resulting in less efficient test case generation. Remarkably, the other reason was discrepancies between the CHC encoding and the actual semantics of the contracts.

SolTG was also evaluated on multiple industrial Ethereum contracts (e.g., `Weth.sol`, `ERC20.sol`, `VestingWallet.sol`), and it exhibited performance similar to that on

the SolCMC benchmarks. For example, for `Weth.sol`[4], SolTG was able to produce tests that achieve 82% line, 83% function, and 75% branch coverage in 20 seconds. Interestingly, experimentation with industrial cases revealed a discrepancy between the CHC encoding and the compiler output: some function calls in Solidity have inherent properties, and for one of these calls these properties are missing in the CHCs. This issue was reported to the developers of SolCMC[5].

Comparison with other tools. The evaluation does not include a quantitative comparison with other test generators for Solidity, since the existing tools are not directly comparable to SolTG. Among the tools considered, only fuzzers could be executed on the benchmarks. The output of a fuzzer, however, is not a test suite in the sense of SolTG: the test cases it produced had to be manually rewritten as Foundry tests before their coverage could be measured in the same way as for the tests generated by SolTG. Moreover, the coverage achieved by fuzzing is not comparable to the coverage of an exhaustive enumeration. Fuzzers explore the input space randomly, so the coverage they reach depends on the random seed and on the length of the fuzzing campaign, and they can easily miss specific program locations, e.g., branches guarded by narrow conditions such as an equality with a particular value. SolTG, in contrast, targets every branch of every function with a symbolic reachability query: it either produces a test that reaches the branch or establishes that no sequence of calls of the considered length reaches it. A coverage comparison would therefore mostly reflect the fuzzing budget rather than the quality of the generated tests. For these reasons, the evaluation focuses on the coverage that SolTG achieves on its own and on the time it needs to achieve it.

Overall, the results demonstrate the practical value of SolTG in the contract development process.

5.4 Limitations

The current version of SolTG has two main limitations.

Loops. SolTG does not support contracts whose functions contain loops. The enumeration of contract behaviors unrolls the CHC representation of the function summaries, which is only finite for loop-free function bodies. For example, no tests can currently be generated for a function such as

```
1 function total() public view returns (uint s) {
2     for (uint i = 0; i < n; i++) s += vals[i];
```

[4] https://etherscan.io/token/0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2.
[5] https://github.com/ethereum/solidity/issues/14814.

```
3 }
```

which iterates over a storage array.

Deep test cases. Test cases are generated by enumerating sequences of function calls of increasing length, and the algorithm checks all possible combinations of function calls. For a contract with f public functions, there are f^d call sequences of length d, and the number of control-flow paths that must be explored grows further with the branches inside each function. For example, a contract with 5 public functions already has $5^4 = 625$ call sequences of length 4. Consequently, branches that become reachable only after a long, specific sequence of calls, such as a function that can only be executed after a contract has been initialized, funded, and moved through several states, may not be covered within the timeout.

The future work in Section 5.6 addresses both limitations: loop analysis for the first one, and commutativity checking, function invariants, and the combination with fuzzing for the second.

5.5 Related Work

The principles of the CHC encoding of smart contracts that SolTG relies on originate from [139]. Previously, they enabled the development of the SolCMC model checker built into the Solidity compiler. SolCMC, however, is not designed for generating test cases for each branch of each function of a given contract, which requires an exhaustive enumeration of path conditions. On the other hand, SolTG does not rely on external CHC solvers and has its own approach to achieve the enumeration. An alternative framework for automated reasoning over Solidity contracts, SmartACE, uses an existing infrastructure for LLVM-based model checking, symbolic execution, or fuzzing [57]. As a result, it does not generate any tests for the original Solidity code, and the generated reports are based on testing of a semantically different LLVM binary. A notable benefit of SolTG over SmartACE is that the former guarantees that behavior demonstrated by the tests exactly corresponds to the actual blockchain behavior. Another notable tool, VerX [155], is designed to prove functional properties of smart contracts. Although it follows a different route than SolTG, the awareness of the discovered properties can help SolTG to accelerate the test case generation in the future.

Fuzzing [99] is the most widely used approach to automated testing of smart contracts, e.g., in Echidna [104] and in the fuzzer integrated into Foundry. Fuzzers execute the contract on large numbers of generated inputs and call sequences, which makes them fast and scalable and lets them explore long sequences of function calls cheaply. However, they give no guarantee of reaching a particular program location,

and they struggle with branches guarded by narrow conditions. SolTG has the opposite profile: its symbolic queries reach such branches directly, but the number of call sequences it enumerates grows exponentially with their length. The two approaches are therefore complementary rather than competing (see Section 5.6).

There exist multiple SMT-based test generation tools, such as KALI [39] and CAVITEST [154] for Java, FuseBMC [2], Symbiotic [58], and KLEE [56] for C, and some others. Many of them produce test cases after some communication with an SMT solver, but none of them is based on CHCs. The closest tool to SolTG is a CHC-based test case generator, Horntinuum, which is, however, not tailored for smart contracts. Horntinuum alternates invariant generation and test case generation, and it accelerates the enumeration process by exploiting the invariants discovered so far. In the future, this strategy could be adopted in SolTG as well.

5.6 Future Work

SolTG offers several directions for future work.

A particularly promising direction is the combination of SolTG with fuzzing, following the idea of whitebox and hybrid fuzzing [100]. The combination can work in both directions. First, the tests generated by SolTG can serve as seeds for a fuzzer: they already reach branches that random inputs are unlikely to hit, and the fuzzer can then mutate them cheaply to explore the behaviors around those branches. Second, a fuzzer can be run first to cover the easy parts of a contract, and SolTG can then target only the branches that fuzzing left uncovered with symbolic reachability queries. The contract states reached by the fuzzer after long call sequences could also be used as starting points for these queries, reducing the exponential cost of enumerating call sequences in SolTG. Such a combination would help determine where symbolic reasoning brings the most benefit, and would broaden the practical impact of the tool.

One direction is to explore commutativity checking of contract functions. One of the main weaknesses of the current technique is the exponential blow-up in complexity as the number of function calls in a test case grows. A commutativity check would eliminate a significant number of tests whenever the order of function calls does not influence the final state of the system. Reducing the number of test cases considered would significantly speed up test generation.

Another possible extension is loop analysis, allowing SolTG to handle smart contracts that contain loops. Such analysis could be performed with TPA or BMC under a hard bound on the number of iterations, which is natural in Solidity: the gas mechanism limits the number of instructions executed during a single function call. This would extend the tool to the broader domain of industrial smart contracts.

Finally, function invariants could be constructed to replace the explicit exploration of a function with its invariant. This would be especially beneficial for tests with multiple function calls: every call except the last could be replaced by its invariant, abstracting the system and yielding faster test-generation queries.

5.7 Conclusion

This chapter presents SolTG, a fully automated Solidity test generator based on CHC encoding capable of generating tests for industry-grade smart contracts. SolTG analyzes the CHC-system provided by SolCMC and compiles test cases as a result of exhaustive enumeration of contract behavior and SMT models, thus avoiding fuzzing. The compiled tests are supported by the widely used Foundry framework. The evaluation demonstrates that the tool is effective in generating tests on a range of Solidity contracts and can be fully integrated into the development process. These results have been presented and published in CAV 2024 [50].

Chapter 6

Flexible SMT Solving for Safety Verification

The success of modern tools for determining the satisfiability of propositional and first-order logic formulae using SAT and SMT solving can be partially attributed to the combination of the search for a satisfying assignment with the search for a refutation of satisfiability [151]. This idea carries over to safety verification, where many tools combine a search for a safe, inductive invariant with a search for a counterexample. This type of safety verification requires tight integration with one or more SAT or SMT solvers. A broad range of tools rely on artefacts produced by the SMT solver, such as satisfying assignments and interpolants [5]. Modern SMT solving might suffer from a lack of diversity in solving strategies, as SMT solvers tend to optimize for one specific parameter — solving performance. However, this might not be optimal for the purposes of safety verification. Difficult verification instances, particularly those requiring an invariant, can be very hard to solve, and the model checker may fail to find a proof even when one exists.

This chapter addresses this problem by proposing a heuristic for SMT solving, called PickyCDCL, that is sufficiently efficient to solve existing benchmarks while remaining orthogonal to standard branching approaches. PickyCDCL is a new Conflict-Driven Clause Learning (CDCL)-based decision procedure with flexible, multi-literal lookahead selection. PickyCDCL combines lookahead-style search [112] with the Variable State Independent Decaying Sum (VSIDS) branching heuristic.

PickyCDCL aims to strike a balance between the speed of the VSIDS heuristic and the careful literal selection of lookahead search. Rather than relying solely on the literal scoring provided by VSIDS, PickyCDCL selects, at each branching decision, a tunable number δ of literals with the best VSIDS scores and performs a lookahead procedure over these selected literals to pick the one to propagate. The use of CDCL

together with the consideration of multiple literals at each decision results in a more detailed search and in additional learned clauses, which may in turn yield artefacts better suited to verification purposes (for example, higher-quality interpolants from interpolating SMT solvers). This chapter explores the impact of this potentially slower but more careful approach on the production of unsatisfiability proofs and interpolants. This contribution was presented and published in [49]. Some of the concepts used in this chapter were introduced in Chapters 1 and 2.

6.1 Preliminaries

SMT solvers have proven to be particularly efficient in solving ground, multi-sorted first-order formulae with arithmetic. The input instance is interpreted as an arbitrary propositional formula over atoms that have Boolean values but may represent constraints in different theories, such as inequalities over arithmetic expressions and equalities over uninterpreted functions defined over uninterpreted sorts. The idea in SMT solving is to find a satisfying assignment or prove the absence of one by combining a solver for propositional satisfiability with solvers for conjunctions of theory equalities and inequalities.

When the theory structure is ignored, the formula can be solved using a SAT solver. The satisfying assignment found by the SAT solver represents a conjunction of equalities and inequalities appearing inside the atoms in the original formula. Under the theory interpretation, this conjunction might still be unsatisfiable, since the SAT solver could assign truth values to the equalities or inequalities that are inconsistent in the underlying theory. To overcome this problem, the theory solvers are queried for the consistency of an assignment proposed by the SAT solver. In the case of unsatisfiability, the theory solvers provide an explanation, expressed as a disjunction of (negated) theory atoms. The theory solver may also be unable to determine satisfiability, in which case it provides new atoms for the SAT solver to branch on.

Conflict-Driven Clause Learning. CDCL is the most popular technique used in modern SAT and SMT solvers. Modern SAT solvers work by gradually expanding a decision stack, assigning truth to the atoms of the instance. The stack can be extended either by branching on values of atoms or by unit propagation. The former is a heuristic process, whereas the latter works by finding clauses in which, under the current truth assignment, exactly one literal is unassigned and all others are assigned false, and then assigning that remaining literal to true. Unit propagation may terminate in a conflict, in which some literal would need to be assigned both true and false simultaneously. This triggers a conflict analysis, which aims to find a point in the truth assignment to

which the solver should be backtracked to resolve the conflict. The analysis is a limited form of resolution and results in a *learned clause*, which is guaranteed to have all but one literal assigned false and the remaining literal unknown at the backtrack point. Pseudocode for the CDCL procedure is given in Algorithm 9. The algorithm picks a literal for branching using some heuristic (the call to BestLit) and runs unit propagation (Propagate). If propagation ends in a conflict, either directly or as reported by a theory solver, the solver then adds a learned clause in line 5 and backtracks to a previous state in the decision stack.

Heuristics. The branching heuristic of an SMT solver has a big impact on solving efficiency. Literals are typically scored using an *activity-based heuristic* such as VSIDS, where atoms that appear often in resolution receive higher scores. VSIDS is one of the most popular heuristics for SMT solving. VSIDS keeps track of the literals that appeared in conflicts, and the number of times these literals were present in the conflict clauses. This heuristic picks for propagation the literal that has been the most present in the conflict clauses first. This is combined with an approach that tries to avoid creating immediate conflicts in the theories by preferring a polarity that is not conflicting with the model maintained internally by the theory solver.

An alternative to activity-based heuristics is the *lookahead heuristic*, which attempts to estimate the size of the remaining search space after branching by computing the number of atoms that would remain unassigned if a given atom were assigned. As a result of the theory explanations and the learned clauses, an SMT solver does not have a fixed number of clauses or even atoms. The number of unassigned atoms may nevertheless be used as an (inaccurate) proxy for the remaining search space size [113].

Algorithm 9: Basic CDCL algorithm.

Input : An SMT instance in CNF ϕ
Output: SAT | UNSAT
Data : Decision stack s, literal x_{best}

```
1 while True do
2     while Propagate (s) results in conflict do
3         if the decision level of s is 0 then return UNSAT;
4         c ← analyse the conflict
5         φ ← φ ∧ c
6         Backtrack s until c becomes implying
7     if s ⊇ Var(φ) then return SAT;
8     x_best ← BestLit()
9     s ← s · ({x_best})
```

Model checking using interpolation. Let $X = \{x_1, \ldots, x_n\}$ be a set of variables in a multi-sorted logic and $X' = \{x' \mid x \in X\}$ a copy of these variables obtained by adding a prime to their symbols (used to represent the next-state values of the variables). The core task of safety verification is to prove that it is not possible to reach a state that violates the safety property $Bad(X)$ from the initial states $Init(X)$.

To prove this property, most algorithms attempt to construct an inductive invariant $Inv(X)$, such that $Inv(X) \wedge Bad(X)$ is unsatisfiable. A range of techniques exists for obtaining the invariant Inv directly from the proof of unsatisfiability that an SMT solver produces as a side-product, for example, by computing interpolants. The construction of interpolants depends on SMT solving, which in turn is guided by the branching heuristic. This observation is at the center of the experimental evaluation of the heuristics in this chapter.

6.2 PickyCDCL

The PickyCDCL algorithm is based on the idea of a "more careful" VSIDS, incorporating a lookahead on top of it. PickyCDCL learns more clauses than VSIDS on each decision level, due to the wider scope of its propagations (as it tries to propagate δ variables at every round). Even in the case when no conflicts are encountered, it guides VSIDS, picking the literal with the highest number of unit propagations. The application of lookahead allows for greedy learning of conflict clauses, as at each decision level more literals are propagated and potentially more conflicts are encountered. Algorithm 10 outlines PickyCDCL.

The heuristic for deciding on literals is a modification of the VSIDS approach with two additional steps. The first is updating the VSIDS scores for the literals involved in the conflict analysis, which PickyCDCL does both on line 4 during the "standard" propagation, and on line 15 when a clause is learned in the lookahead phase. The second step implements the procedure that picks the δ best literals with the highest VSIDS scores, instead of selecting just one best literal. The value δ is a parameter of the algorithm. It gives PickyCDCL the flexibility to pick between exhaustive and quick approaches. PickyCDCL runs a lookahead algorithm for the picked literals, choosing the literal that will have the biggest impact on the propagation, i.e., the literal that, when fixed, causes the largest number of unit propagations as its consequence.

The main execution loop propagates literals one by one and tries to find a satisfying literal assignment. Initially, the algorithm propagates the current decision stack. When the propagation results in a conflict, the algorithm handles it. If the solver is at decision level 0, the instance is unsatisfiable. Otherwise, the solver introduces a new clause to the instance, blocking the conflicting assignment, and backtracks to the decision level

Algorithm 10: PickyCDCL algorithm.

Input : An SMT instance in CNF ϕ; search width δ
Output: SAT | UNSAT
Data : Decision stack s, literal x_{best}, map scores : $At(\phi) \to (\mathbb{N})$

```
1  while True do
2      while Propagate (s) results in conflict do
3          if the decision level of s is 0 then return UNSAT;
4          c ← analyse the conflict
5          φ ← φ ∪ {c}
6          Backtrack s until the second-highest decision level in c
7      if s ⊇ Var(φ) then return SAT;
8      x_best ← ⊤
9      scores[x] ← 0 for all x ∈ var(φ) ;              // Reset literal scores
10     lits⃗ ← BestLits(δ)
11     for x ∈ lits⃗ do
12         s ← s · ({x})
13         if Propagate (s) or CheckTheory () results in conflict then
14             if the decision level of s is 0 then return UNSAT;
15             c ← analyse the conflict
16             φ ← φ ∧ c
17             scores[x] ← 0
18         else
19             scores[x] ← la − score(x) ;          // updates lookahead score
20             if scores[x] > scores[x_best] then x_best ← x ;
21         Backtrack s until x becomes undefined
22     s ← s · ({x_best});                          // Best literal is propagated
```

where the conflict is resolved (lines 4–6). At line 7, the algorithm checks whether all variables have been assigned, meaning the instance is satisfiable. The remaining part of the code implements the heuristic that picks the literal for the next propagation.

PickyCDCL starts by resetting the current best literal x_{best} to $\top$ (line 8) and the score of every literal to 0 (line 9). At line 10, the algorithm picks the δ literals with the highest VSIDS scores and stores them in the vector $\vec{lits}$. This parameter can range from $\delta = 1$ to $\delta = nLits$, the total number of literals in the instance. The latter corresponds to running a full lookahead on every branching decision. The former is almost equivalent to the VSIDS heuristic, with a small added overhead because of the second propagation[1]. The parameter δ adds flexibility to the algorithm: it allows the balance between performance (with a smaller δ) and rigor (with a larger δ) to be tuned. δ is provided by the user of the solver and can vary between different instances. Experiments show that the value of δ should be picked carefully. A linear increase of the parameter δ may cause a quadratic growth in complexity, because δ literals are propagated during each iteration of PickyCDCL. In other words, the δ parameter regulates the breadth of the search: decreasing it reduces the selection precision, while increasing it enlarges the exploration space for the literal selection.

PickyCDCL performs test propagations for all the literals in the vector $\vec{lits}$ (δ literals with the highest VSIDS scores). Among these literals, the algorithm identifies the one that leads to the largest number of subsequent unit propagations. If conflicts are encountered during the test propagations, PickyCDCL adds the learned clauses to the instance, but does not stop the lookahead procedure. During each test propagation, PickyCDCL also performs a complete theory check (the call to CheckTheory () at line 12), to validate that the tentative assignment is consistent with the underlying SMT theory, even when the Boolean propagation itself does not yield a conflict. Because of this additional check, even PickyCDCL with $\delta = 1$ diverges from the classical VSIDS heuristic: it catches theory conflicts earlier and, as a result, produces more conflict clauses during the search. In the case of a successful propagation, at line 19, PickyCDCL records the lookahead score for x; the search is then rolled back (line 21) to the state before the test propagation of x.

Overall, the PickyCDCL algorithm can be described as an enhanced CDCL(T) with VSIDS. It incorporates CDCL(T) by allowing clause learning during propagation and expands the VSIDS heuristic by selecting the δ best literals and applying lookahead to them during literal selection. This approach provides more ways to find a solution and varies depending on the value of δ. The PickyCDCL approach learns new clauses greedily, while following the VSIDS heuristic. Compared to the classical VSIDS, Pick-

[1]Propagation of the literal happens twice: at line 1 and at line 12. This double propagation allows PickyCDCL to catch additional conflicts, but it introduces a slight overhead.

YCDCL is expected to produce solutions more suitable for the applications that use interpolation-based SMT solving. This is investigated experimentally through the use of PICKYCDCL in different applications.

6.3 Related Work

This work builds upon the CDCL [166] and DPLL [19, 68] approaches for SAT solving, which were later adapted for use in SMT. The CDCL algorithm is a conflict-driven methodology that operates by backtracking when conflicts are encountered and utilizing the resulting information to generate clauses that restrict the assignment of literals causing the conflicts.

The technique is based on the Variable State Independent Decaying Sum (VSIDS) heuristic, which greatly improves the splitting part of CDCL(T) by quickly selecting literals for splitting based on how frequently they occur in recently learned (conflict) clauses, with scores decaying over time to favor more recent activity. VSIDS is a widely used heuristic in many solvers, resulting in minimal time spent on selecting literals for branching.

Lookahead heuristics for SAT and SMT solving are of key importance to this work. The core idea of a lookahead SAT solver is to propagate all literals and select the one that leads to the most collateral propagations, repeating the process until all literals are propagated [38]. The first lookahead-based solver was developed in 1996 and was a top-performing SAT solver at the time. Lookahead has since been adopted by several major SAT/SMT solvers, such as Z3, where it is used for selecting cubes for splitting in the SAT solver core. Recently, the lookahead algorithm was applied to SMT solving in the OPENSMT solver [118]. This implementation showed improvements over VSIDS on certain benchmark sets, even though it used a significant number of additional propagations, checking all literals at each step, even previously checked ones. This approach was specifically designed for constructing evenly-sized subproblems to partition a search space in parallel solving. This work differs from that approach in that it overcomes its performance bottleneck and does not maintain a DPLL-style tree for parallelization.

The work closest to PICKYCDCL combines and refits SAT/SMT heuristics. Mixing lookahead and VSIDS in SAT solving applications was mentioned in [180] as a means of breaking ties between equal VSIDS scores to increase the performance of a solver. If the scores are the same, lookahead is used on the literals with equivalent scores to pick the literal for propagation. However, this approach did not lead to a speedup in the solving process. Unlike the aforementioned approach, PICKYCDCL uses a limited form of lookahead that runs at each decision level during solving. Lookahead is executed on

a certain number of top-scored VSIDS literals, not necessarily sharing the same score. The goal of this work was to increase the amount of information obtained during solving, while remaining performant in terms of deciding satisfiability. The ultimate goal of PickyCDCL is not to improve the performance of the solver but to produce solving artefacts that carry more information suitable for verification tasks. Experiments with several verification tools show that PickyCDCL achieves this goal.

In recent years, significant efforts have been made in SMT solving on the production of proofs (see [14, 152] for examples) and interpolants (see [125] for a thorough overview). Proofs serve as evidence of the correctness of an unsatisfiability result and have become a crucial aspect of SMT solvers. The quality of the proof depends on the solution path chosen by the solver, and CDCL-based approaches are expected to lead to better proofs as they generate more conflict clauses, providing more information for proof production. More conflict clauses also allow for the creation of more accurate interpolants, as additional constraints can be imposed on the system, reducing the set of possible assignments.

6.4 Evaluation

The PickyCDCL algorithm was implemented on top of the interpolating SMT solver OpenSMT[2] with the production-ready versions of lookahead and VSIDS heuristics. The solver works on quantifier-free logics of UF, LIA, and LRA, which provided a wide range of opportunities for experimentation with the new heuristics. Both the PickyCDCL and VSIDS algorithms were used in the experiments.

This section evaluates the algorithm in the context of interpolation-based model checking and proof production. The main idea of the algorithm was to expand the applicability of SMT solving to better suit verification tasks; the experiments therefore focus on SMT-based safety verification. PickyCDCL was applied to two applications: model checking that uses interpolation for invariant generation (in a CHC solver) and proof production/validation. Additionally, the overhead of running PickyCDCL compared to classical VSIDS was evaluated. All experiments were conducted with optimizations enabled, on a machine with an AMD EPYC 7452 32-core processor and 8x32 GiB of memory. To test PickyCDCL's performance and its application in safety verification, CHC-Competition benchmarks were used. During the experiments, no conflicting results were encountered. Throughout this section, relative differences in running time and size are reported as factors (e.g., "1.3 times slower"), together with the absolute values where they are available.

[2] https://github.com/usi-verification-and-security/opensmt; PickyCDCL is available in the main branch and is enabled with the `:picky` option.

6.4.1 Interpolation-based Algorithms

The effect of the application of the PICKYCDCL algorithm on interpolation-based reasoning was studied in the context of Constrained Horn Clauses solving. The experiments use the GOLEM CHC solver[3], which integrates OPENSMT into its reasoning both for interpolation and solving. Three of GOLEM's verification engines were used: TPA, LAWI, and Spacer, all of which used an OPENSMT solver with PICKYCDCL and VSIDS heuristics as a backend for SMT solving. The following is a comparative analysis of these techniques with PICKYCDCL and VSIDS-based backend SMT solving.

The comparison measures how applying OPENSMT with PICKYCDCL support to GOLEM's TPA, Spacer, and LAWI verification engines affects their verification capabilities. In particular, it compares the number of instances solved by each verifier with OPENSMT's VSIDS and PICKYCDCL core procedures. Furthermore, since PICKYCDCL is flexible in the number of literals used for propagation (defined by parameter δ), three instances of PICKYCDCL were evaluated in all experiments. The evaluated configurations were PICKYCDCL with $\delta = 1$ (corresponding conceptually to the classical VSIDS algorithm), PICKYCDCL with $\delta = nLits$ (corresponding conceptually to the classical lookahead), and the mixed PICKYCDCL with δ set to 10 (the choice of "10" was based on an empirical evaluation of various benchmarks[4]).

An evaluation was executed on the LRA-TS (498 benchmarks) and LIA-lin (585 benchmarks) benchmark sets from CHC-COMP [69], the benchmarks for which GOLEM had highly competitive performance results. The experiments used a timeout of 1800 seconds. The results are shown in Table 6.1. The number in parentheses is the count of uniquely verified instances, compared to the default SMT-solver engine, which is based on the VSIDS heuristic (listed first for each tool in the table). The goal was to determine whether using the PICKYCDCL algorithm, even though it may be slower than the core technique, would allow the verification engines to solve CHC instances that were not previously solvable using VSIDS. The results also include two further lookahead-based approaches, both implemented in the OPENSMT solver: the DPLL-based lookahead [118] (Lookahead DPLL) and the tiebreaker approach [180] (Tiebreaker Lookahead), in which lookahead is used only to break ties between literals with equal VSIDS scores.

Overall, the comparative experimentation demonstrated that in most cases when the PICKYCDCL algorithm was used, GOLEM was able to solve additional instances that could not be solved by the core procedure alone. As expected, the core technique is usually able to solve more instances within the same timeout; however, the delay

[3] https://github.com/usi-verification-and-security/golem.

[4] Across a wide range of individual experiments, the δ parameter in the range 10–20 was found to provide the best CHC-solving performance for GOLEM with the PICKYCDCL algorithm.

Table 6.1. Golem results comparing various interpolation-based verification algorithms based on either VSIDS or PickyCDCL procedures of OpenSMT.

Algorithms	LIA-lin	LRA-TS
Number of benchmarks	585	498
TPA		
VSIDS	340	214
PickyCDCL with $\delta = 1$	338 (5)	202 (3)
PickyCDCL with $\delta = 10$	334 (7)	200 (6)
PickyCDCL with $\delta = nLits$	313 (2)	186 (5)
Lookahead DPLL	284 (1)	162
Tiebreaker Lookahead	321 (5)	192 (5)
Spacer		
VSIDS	381	270
PickyCDCL with $\delta = 1$	379 (2)	270
PickyCDCL with $\delta = 10$	366 (2)	272 (2)
PickyCDCL with $\delta = nLits$	340	249
Lookahead DPLL	302	221
Tiebreaker Lookahead	372 (1)	265 (1)
LAWI		
VSIDS	322	364
PickyCDCL with $\delta = 1$	315 (2)	358 (8)
PickyCDCL with $\delta = 10$	312 (3)	325 (10)
PickyCDCL with $\delta = nLits$	294 (3)	237
Lookahead DPLL	265 (2)	208
Tiebreaker Lookahead	308 (2)	279 (3)

Table 6.2. Number of instances solved by virtual portfolios of Golem's TPA engine with different random seeds and SMT backends (3 engines per portfolio).

Virtual portfolio composition	LIA-lin	LRA-TS
3 VSIDS-based SMT solvers	345	223
2 VSIDS and 1 PickyCDCL-based SMT solvers	352	225
1 VSIDS and 2 PickyCDCL-based SMT solvers	354	230
3 PickyCDCL-based SMT solvers	343	220

introduced by the PickyCDCL procedure (see the overhead study below) corresponds only to a small reduction in the number of benchmarks solved. PickyCDCL with $\delta = 1$ is able to solve unique instances compared to VSIDS due to the divergence in the solving process explained in Section 6.2. Interestingly, for Spacer on LRA-TS, PickyCDCL with $\delta = 10$ even solved more instances than the VSIDS procedure (272 vs. 270). Overall, the results show that the mixed PickyCDCL approach is not only more flexible in its search, since it is able to solve previously unsolved benchmarks, but also retains much of the performance efficiency of the classical algorithms, remaining comparable to the other techniques used in the evaluation.

Robustness to random seeds. To illustrate that the effect of using the PickyCDCL algorithm in Golem is not random, a further experiment was conducted with Golem, using the same set of LIA-lin and LRA-TS benchmarks as in the experiments above. Three instances of the VSIDS-based solver and three instances of the PickyCDCL-based solver were created, all with different random seeds. They were then run inside the Golem CHC solver to produce a virtual portfolio of solved instances, where each portfolio consists of three Golem instances with different mixtures of the SMT backends. Table 6.2 summarizes the results of running the virtual portfolios. Notably, the mixed portfolios that combine VSIDS and PickyCDCL are able to solve more instances overall. They systematically outperform the portfolio based solely on the VSIDS approach, despite pure VSIDS being potentially more efficient.

Some benchmarks (chosen randomly) that Golem uniquely solved using PickyCDCL were also examined closely and rerun with ten instances of the VSIDS-based Golem with different random seeds. For example, *chc-LIA-Lin_325* was not solved by any of the random-seed Golem instances. Another example is the instance *chc-LIA-Lin_035*, which was solved by the VSIDS-based Golem only 2 times out of 10. This shows that PickyCDCL-based Golem can uniquely solve certain instances, independently of the random seed used. Overall, the experiments indicate that the lookahead-inspired mixed approach can lead the solver to solutions that are better suited to

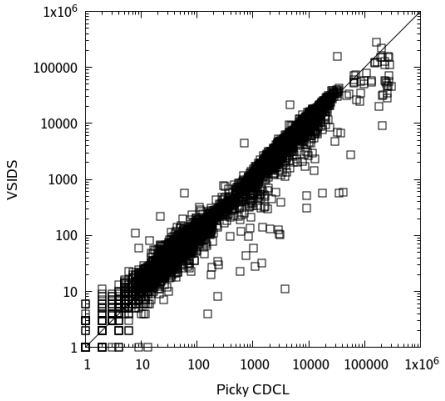


Figure 6.1. A comparison of proof size for the VSIDS approach vs. PickyCDCL. (In the number of unique lemmas used.)

verification tasks, which makes PickyCDCL a valuable component of a portfolio for interpolation-based CHC solving.

6.4.2 Proof Production

Another set of experiments was conducted with OpenSMT's proof production and proof validation engine. Proofs are an important part of any modern SAT/SMT solver because they make it possible to validate the correctness of unsatisfiability results, confirming that the solver did not err. OpenSMT supports proof production using the VSIDS SMT solving procedure. For this study, OpenSMT was adapted to support proof production using the PickyCDCL algorithm instead. OpenSMT produces proofs in the DRAT format; the effect of using PickyCDCL compared to the VSIDS approach on proof size and proof validation time was evaluated.

The proof size was estimated by measuring the number of employed lemmas in the corresponding formulae. A series of experiments was conducted on `QF_UF` benchmarks taken from SMT-COMP. Overall, 1752 benchmarks were evaluated with a 1200-second time limit, all of which demonstrated a similar trend. The results show that the proofs produced by OpenSMT with the PickyCDCL backend contain, on average, 1.15 times as many lemmas as proofs produced by the VSIDS-based solver. This can be attributed to the fact that the PickyCDCL procedure learns clauses greedily during solving, thereby expanding the pool of lemmas available for the proofs. The results for OpenSMT based on VSIDS and PickyCDCL are shown in Figure 6.1.

The time it took to validate proofs produced from the `QF_UF` benchmarks solved by both the VSIDS-based and PickyCDCL-based solvers was also compared, using a 1000-second timeout. The PickyCDCL-based proofs were expected to be at least as

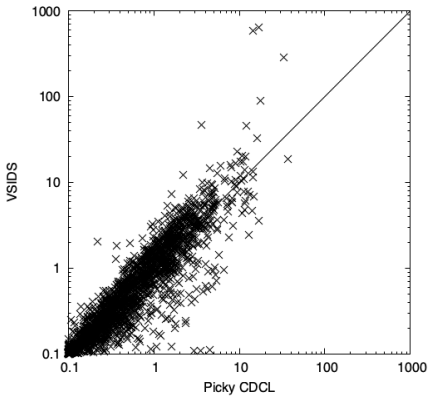


Figure 6.2. A comparison of the validation time for VSIDS-built proofs against the PickyCDCL-built proofs. (Time comparison in seconds.)

easy to validate as the VSIDS-based ones, thanks to the additional clauses learned from the detected conflicts. The DRAT-trim [111, 178] tool was used to validate the proof results. The results can be seen in Figure 6.2.

The graph data shows that the proof validation process was faster for proofs produced by the PickyCDCL-based solver than for those produced by the VSIDS-based solver. The Y-axis on the graph represents the time taken to validate the VSIDS-produced proof, and the X-axis represents the time taken to validate the PickyCDCL proof. In total, validating the PickyCDCL proofs took 2475.49 seconds, whereas validating the VSIDS proofs took 4203.87 seconds, i.e., PickyCDCL proofs were validated 1.7 times faster. This implies that PickyCDCL was able to create proofs that are easier to validate[5]. Therefore, PickyCDCL can be used in a portfolio for proof-based interpolation in model checking with a positive effect.

6.4.3 Overhead Evaluation

The mixed approach proposed for the flexible literal selection of the PickyCDCL CDCL procedure naturally requires more computational resources than the core VSIDS approach. The overhead of the combined approach for OpenSMT was studied on the SMT-COMP benchmarks for `QF_LRA`. The comparison used the PickyCDCL implementation with different values of the δ parameter ($\delta = nLits$, $\delta = 10$, and $\delta = 1$). The timeout for SMT solving was set to 1200 seconds. Overall, increasing δ was observed to lead to a more significant performance overhead; for $\delta = nLits$, the slowdown compared to VSIDS was the same as for the full lookahead approach. This happens due to

[5]The solver may be selecting suboptimal clauses during its solving process, which could be the cause of points below the diagonal.

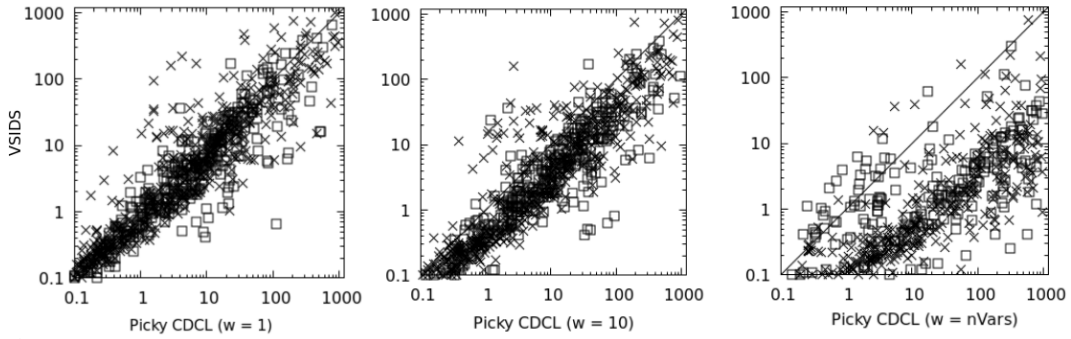


Figure 6.3. Overhead evaluation for OpenSMT with PickyCDCL-based vs. VSIDS-based solving for various depths of literal selection. (Time comparison in seconds.)

the increased number of literal propagations as δ increases (because more candidate literals are considered at each step).

Figure 6.3 illustrates the comparison of PickyCDCL with VSIDS for the cases of $\delta = 1$, $\delta = 10$, and $\delta = nLits$. Crosses on the graph denote SAT results, and squares denote UNSAT results. Noticeably, PickyCDCL with $\delta = 1$ is on average 1.3 times slower than VSIDS, due to the reasons explained in Section 6.2. PickyCDCL with $\delta = 10$ is on average 1.81 times slower than VSIDS, even though it propagates ten times more literals at each decision round and finds 2-3 times more conflicts than the classical VSIDS approach. As such, the algorithm becomes significantly faster than the lookahead approach, especially with a smaller δ, while still finding significantly more information about the instance (learned clauses) than VSIDS.

6.4.4 Lookahead Comparison

The PickyCDCL heuristic was developed as a mixed approach combining VSIDS and lookahead strategies. Both of these techniques were implemented in OpenSMT prior to the development of PickyCDCL. Lookahead in OpenSMT is based on the DPLL procedure, with the main goal of splitting SMT solving for parallel reasoning. During the development of the DPLL-based lookahead in parallel OpenSMT, solving performance was secondary, because it was developed for efficient distribution of reasoning. As a result, lookahead is slower compared to the standard VSIDS heuristic. Since both lookahead and PickyCDCL are implemented in the same tool, their performance can be compared directly. The two implementations are orthogonal: the lookahead procedure is based on DPLL, while PickyCDCL is based on a pure CDCL engine. The PickyCDCL algorithm was executed with different values of δ ($\delta = nLits$, $\delta = 10$, and $\delta = 1$). The results of the PickyCDCL and lookahead comparison are presented

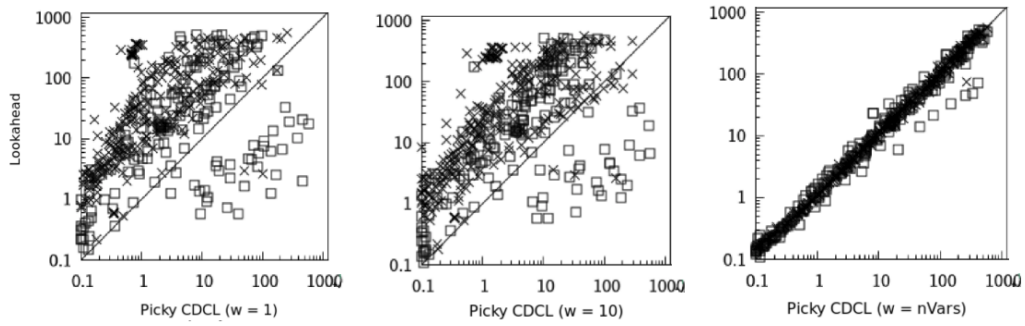


Figure 6.4. PickyCDCL-based vs. lookahead-based OpenSMT performance comparison for various depths of literal selection. (Time comparison in seconds.)

in Figure 6.4.

As expected, PICKYCDCL with smaller δ turned out to be significantly faster than the lookahead approach; for example, PICKYCDCL with $\delta = 10$ is 8 times faster than lookahead. For $\delta = nLits$, however, the PICKYCDCL approach turned out to be as fast as the lookahead technique. Therefore, performance-wise, PICKYCDCL can outperform the lookahead technique and does not underperform it even at the maximal value of δ.

6.5 Limitations

The main limitation of PICKYCDCL is its performance compared to VSIDS. Even with $\delta = 1$, PICKYCDCL is on average 1.3 times slower than VSIDS, and with $\delta = 10$ it is 1.81 times slower. As a result, a verifier that uses PICKYCDCL instead of VSIDS solves fewer instances overall. For example, GOLEM's TPA engine on the LIA-lin benchmarks solves 334 instances with PICKYCDCL ($\delta = 10$), 7 of which are not solved with VSIDS, but it fails on 13 instances that are solved with VSIDS (Table 6.1). PICKYCDCL is therefore most valuable as a component of a portfolio rather than as a replacement for VSIDS.

Moreover, it is not yet known which properties of an instance make PICKYCDCL beneficial. The instances that PICKYCDCL solves uniquely could not be characterized in advance, so it is currently not possible to predict for which instances PICKYCDCL should be preferred, or which value of δ to choose. Understanding the mechanism behind the effect, e.g., by analyzing how PICKYCDCL changes the learned clauses, the structure of the proofs, and the resulting interpolants, is discussed in the next section.

6.6 Future Work

As a future direction, a more randomized approach can be explored. Running the lookahead check at every decision of the heuristic is not very efficient, as it requires a significant number of propagations. It is not only costly but also redundant, since the lookahead check is rerun for literals that were already evaluated in the previous iteration. Instead, the lookahead scores of the top δ literals could be cached and refreshed only periodically, rather than at every decision, to keep the propagation scores approximately up to date. This would significantly reduce the number of additional propagations without sacrificing the thoroughness of the check.

Another idea is a self-tuning δ. Instead of fixing δ when the SMT solver is initialized, the solver could adjust it dynamically during solving. When the lookahead consistently agrees with the VSIDS choice, δ can be decreased toward 1; when the lookahead frequently disagrees with VSIDS, the pool of candidate literals is enlarged. This would make the framework more flexible and reduce its overhead.

6.7 Conclusions

Most research projects in SAT/SMT solving prioritize performance when developing and improving solvers. Faster execution times and the capacity to analyze complex tasks can greatly extend what these solvers are able to accomplish. However, in the pursuit of speed, practical applications of the tools can be neglected. This chapter investigated how lookahead-inspired SMT solving, when made sufficiently efficient and integrated into a CDCL-based SMT core, can be a valuable component in a portfolio for proof-based interpolation in model checking.

This chapter implemented the algorithmic idea called PickyCDCL in the OpenSMT solver and demonstrated its effectiveness when applied to interpolation-based algorithms (using Golem) and SMT proof validation. Overall, this approach demonstrates that more careful methods can yield better outcomes for specific cases. The extensive experimentation with the PickyCDCL-based solver showed that it was able to solve unique CHC benchmark instances that were not solved by classical VSIDS and that it produced proofs that were easier to validate. The study of the overhead cost of the procedure shows that it can be tuned via the δ parameter, trading off overhead against the thoroughness of the search.

The results of this study demonstrate that while the pursuit of the most performant approach is a powerful method, practical outcomes often benefit from a slower, steadier approach. For practical applications, it is thus beneficial to include a few solving algorithms that can cover a broader portion of the solution space, increasing the

tool's overall solving capabilities. This contribution was published and presented at the VSTTE 2023 workshop [49].

Chapter 7

Conclusions

This thesis targets the automated verification of software: the ability to formally prove that a given program is correct with respect to given specification, or to find a counterexample that violates them. To prove the properties of a program, the thesis uses intermediate formal representations, particularly transition systems and Constrained Horn Clauses (CHCs). Transition systems allow simple reasoning about the program, while CHCs make it possible to maintain and leverage the program's structural information.

The research presented in this thesis tackles the complexity of program verification from several angles. It presents solutions that demonstrate how verification challenges can be addressed at different, often interconnected, layers. The techniques used within the verification process (such as SAT/SMT solving), the verification algorithms themselves, and their applications are mutually dependent and benefit from targeted optimizations and improvements. Based on this idea, the thesis addresses the problem of automated software verification through a three-layered approach, namely the foundational layer, the algorithmic layer, and the application layer. The foundational layer is centered around SAT/SMT solving. It is responsible for solving logical formulae and producing models and interpolants during the verification process. Most verification techniques rely on SMT solving as a black box; however, truly efficient verification can be achieved only if solving is optimized and integrated with the specific verification technique. The algorithmic layer is responsible for the actual verification of the program. At this layer, the thesis investigates the importance and benefits of software structural information for verification and the substantial effect of modular approach, which significantly speeds up the verification process. Additionally, this layer investigates the combined mutually beneficial termination and nontermination analysis of programs. Finally, the application layer utilizes safety checking for problems different from safety verification. Verification can be actively and efficiently used for automated

test generation and other applications, allowing for the development of more reliable software.

Chapter 3 presented a new structure-aware approach for the efficient verification of complex software systems. The introduced verification approaches target software with multiple loops, nested loops, and complex control flow, which are challenging for state-of-the-art verification techniques. The two newly designed verification techniques described in that chapter utilize software structural information and consider the safety problem in a modular manner, which speeds up the verification process. The techniques leverage the TPA verification approach to quickly find deep loop counterexamples and to generate transition invariants for loops. The experimental evaluation demonstrated that these techniques are more efficient than state-of-the-art approaches for the verification of complex software systems. The results of the research presented in this chapter were published at the FM 2024 [47] and iFM 2025 [48] conferences.

Chapter 4 presented a new approach for termination and nontermination analysis of programs, using safety checking techniques as its core. Safety verification is used to construct invariants, from which recurrent sets are extracted to prove nontermination. Additionally, the counterexamples found during safety verification are used to construct transition invariants via interpolation, which makes it possible to prove termination of the program and to guide the nontermination analysis. The experimental evaluation demonstrated that the approach is competitive with state-of-the-art techniques for proving termination and nontermination, and is able to solve instances that had never been solved before.

Chapter 5 introduced SolTG, a new tool for automated test generation for Solidity smart contracts. The tool is based on a CHC encoding of the smart contracts and uses automated verification techniques to generate executable unit tests for them. A bounded approach was used, generating increasingly deep test cases until the program reaches full line and branch coverage. The evaluation of the tool demonstrated that it is capable of generating test cases for industry-grade smart contracts and that it is easy to integrate into the development process. The results of the research presented in this chapter were published at the CAV 2024 [50] conference.

Finally, Chapter 6 investigated the effect of SMT-solving heuristics on the verification process and interpolant production. It introduced a new SMT-solving procedure, PickyCDCL, that combines the standard VSIDS heuristic of the CDCL procedure with the lookahead heuristic, and evaluated its effect on verification and proof production. As a result, it was demonstrated that slowing down the SMT-solving process does not necessarily slow down the verification process, and that in some cases changing the heuristic allows previously unsolvable instances to be solved. The construction of interpolants critically affects the verification process and is as important as the speed of the SMT-solving. Additionally, the effect of the heuristic on the production of unsatisfi-

ability proofs was demonstrated. The results of the research presented in this chapter were published at the VSTTE 2023 [49] workshop.

In summary, the work in this thesis has advanced research in formal methods. We made contributions across different layers of software verification: researching the influence of SMT-solving heuristics on the verification process, investigating modular and structure-aware techniques for automated software verification and termination analysis, and researching applications of verification for automated test generation. This work demonstrates the value of viewing software verification as a combination of different techniques, rather than treating separate layers (such as safety verification or SMT solving) as black boxes. Choosing the right verification approach or SMT-solving method can significantly help in solving the problem at hand.

Future work Several directions for future work emerge from the results of this thesis. For the verification approaches of Chapter 3, three extensions stand out. Because the analysis is modular, the paths it explores through the cutpoint graph could be explored in parallel: a multithreaded implementation in which threads share discovered information. Reversing the search direction with a PDR-style backward exploration, from the unsafe states toward the initial states, could further improve performance, particularly when run as a portfolio alongside the original forward technique. Incremental verification is a third direction. Verification produces artefacts which could be reused after a localized change to the program, recomputing only what the change invalidates and greatly speeding up re-verification.

The termination analysis of Chapter 4 would benefit from the structure-aware treatment: splitting the transition system into individual loops, analyzed in isolation, would speed up the analysis and simplify both the interpolation and the invariant search that currently dominate its cost. Its well-foundedness check could also be strengthened by admitting lexicographic, multiphase, or non-linear ranking arguments in place of purely linear ones; since the algorithm does not depend on the ranking-function synthesis method, richer arguments would let more disjuncts, and hence more candidate transition invariants, be accepted.

SoLTG (Chapter 5) offers several research opportunities. Checking the commutativity of contract functions would remove many redundant tests whenever the order of function calls does not affect the final state, mitigating the exponential growth in the number of generated test cases. Supporting contracts that contain loops would broaden the tool's applicability to industrial contracts. Constructing function invariants is a third direction: in a multi-call test, every call but the last could be replaced by its invariant, abstracting the system and speeding up the test-generation queries.

PickyCDCL (Chapter 6) can be made lighter in two complementary ways. Calling the lookahead procedure less frequently would reduce the overhead while retaining

most of the thoroughness of the check. Specifically the scores of the top candidate literals can be cached and refreshed only periodically. In addition, the number of literals over which lookahead is run could be tuned automatically during solving: enlarged when PICKYCDCL's choice diverges from VSIDS, and reduced toward one when the two agree.

A common thread across these directions is that the modular, structure-aware design of the techniques leaves clear room for parallelization and for reusing intermediate results across analyses.

Bibliography

[1] Aho, A. V., Sethi, R. and Ullman, J. D. [1986]. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley series in computer science / World student series edition, Addison-Wesley.
URL: *https://www.worldcat.org/oclc/12285707*

[2] Alshmrany, K. M., Menezes, R. S., Gadelha, M. R. and Cordeiro, L. C. [2020]. Fusebmc: A white-box fuzzer for finding security vulnerabilities in C programs, *CoRR* .
URL: *https://arxiv.org/abs/2012.11223*

[3] Alt, L., Asadi, S., Chockler, H., Even-Mendoza, K., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2017]. Hifrog: Smt-based function summarization for software verification, *in* A. Legay and T. Margaria (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II*, Vol. 10206 of *Lecture Notes in Computer Science*, pp. 207–213.
URL: *https://doi.org/10.1007/978-3-662-54580-5_12*

[4] Alt, L., Blicha, M., Hyvärinen, A. E. J. and Sharygina, N. [2022]. Solcmc: Solidity compiler's model checker, *in* S. Shoham and Y. Vizel (eds), *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, Vol. 13371 of *Lecture Notes in Computer Science*, Springer, pp. 325–338.
URL: *https://doi.org/10.1007/978-3-031-13185-1_16*

[5] Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2015]. A proof-sensitive approach for small propositional interpolants, *VSTTE 2015*, Vol. 9593 of *LNCS*, Springer, pp. 1–18.

[6] Alt, L., Hyvärinen, A. E. J. and Sharygina, N. [2017]. LRA interpolants from no man's land, *in* O. Strichman and R. Tzoref-Brill (eds), *Hardware and Soft-*

ware: Verification and Testing - 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings, Vol. 10629 of *Lecture Notes in Computer Science*, Springer, pp. 195–210.
URL: *https://doi.org/10.1007/978-3-319-70389-3_13*

[7] Armando, A., Mantovani, J. and Platania, L. [2006]. Bounded model checking of software using SMT solvers instead of SAT solvers, *in* A. Valmari (ed.), *Model Checking Software, 13th International SPIN Workshop, Vienna, Austria, March 30 - April 1, 2006, Proceedings*, Vol. 3925 of *Lecture Notes in Computer Science*, Springer, pp. 146–162.
URL: *https://doi.org/10.1007/11691617_9*

[8] Asadi, S., Blicha, M., Hyvärinen, A. E. J., Fedyukovich, G. and Sharygina, N. [2020]. Farkas-based tree interpolation, *in* D. Pichardie and M. Sighireanu (eds), *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, Vol. 12389 of *Lecture Notes in Computer Science*, Springer, pp. 357–379.
URL: *https://doi.org/10.1007/978-3-030-65474-0_16*

[9] Atzei, N., Bartoletti, M. and Cimoli, T. [2017]. A survey of attacks on Ethereum smart contracts (SoK), *Principles of Security and Trust, POST 2017*, Lecture Notes in Computer Science, Springer, pp. 164–186.

[10] Bajczi, L., Mondok, M. and Molnár, V. [2025]. Theta as a horn solver, *in* E. D. Angelis and F. Frohn (eds), *Proceedings of the 12th Workshop on Horn Clauses for Verification and Synthesis, HCVS@CAV 2025, Zagreb, Croatia, 22 July 2025*, Vol. 434 of *EPTCS*, pp. 27–39.
URL: *https://doi.org/10.4204/EPTCS.434.5*

[11] Baldoni, R., Coppa, E., D'Elia, D. C., Demetrescu, C. and Finocchi, I. [2018]. A survey of symbolic execution techniques, *ACM Comput. Surv.* **51**(3): 50:1–50:39.
URL: *https://doi.org/10.1145/3182657*

[12] Ball, T., Majumdar, R., Millstein, T. D. and Rajamani, S. K. [2001]. Automatic predicate abstraction of C programs, *in* M. Burke and M. L. Soffa (eds), *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, ACM, pp. 203–213.
URL: *https://doi.org/10.1145/378795.378846*

[13] Barbosa, H., Barrett, C. W., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M.,

Reynolds, A., Sheng, Y., Tinelli, C. and Zohar, Y. [2022]. cvc5: A versatile and industrial-strength SMT solver, *in* D. Fisman and G. Rosu (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022*, Vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 415–442.
URL: *https://doi.org/10.1007/978-3-030-99524-9_24*

[14] Barbosa, H., Reynolds, A., Kremer, G., Lachnitt, H., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Viswanathan, A., Viteri, S., Zohar, Y., Tinelli, C. and Barrett, C. W. [2022]. Flexible proof production in an industrial-strength SMT solver, *in* J. Blanchette, L. Kovács and D. Pattinson (eds), *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, Vol. 13385 of *Lecture Notes in Computer Science*, Springer, pp. 15–35.

[15] Bardin, S., Finkel, A., Leroux, J. and Petrucci, L. [2008]. FAST: acceleration from theory to practice, *Int. J. Softw. Tools Technol. Transf.* **10**(5): 401–424.
URL: *https://doi.org/10.1007/s10009-008-0064-3*

[16] Barrett, C., Fontaine, P. and Tinelli, C. [2016]. The satisfiability modulo theories library (SMT-LIB), https://smt-lib.org.

[17] Barrett, C., Stump, A., Tinelli, C. et al. [2010]. The SMT-LIB standard: Version 2.0, *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*, Vol. 13, p. 14.

[18] Barrett, C. W., de Moura, L. M. and Stump, A. [2005]. SMT-COMP: satisfiability modulo theories competition, *in* K. Etessami and S. K. Rajamani (eds), *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, Vol. 3576 of *Lecture Notes in Computer Science*, Springer, pp. 20–23.
URL: *https://doi.org/10.1007/11513988_4*

[19] Barrett, C. W., Nieuwenhuis, R., Oliveras, A. and Tinelli, C. [2006]. Splitting on demand in SAT modulo theories, *in* M. Hermann and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning, 13th International Conference, LPAR 2006, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, Vol. 4246 of *Lecture Notes in Computer Science*, Springer, pp. 512–526.

[20] Barrett, C. W., Sebastiani, R., Seshia, S. A. and Tinelli, C. [2021]. Satisfiability modulo theories, *in* A. Biere, M. Heule, H. van Maaren and T. Walsh (eds), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence*

and Applications, IOS Press, pp. 1267–1329.
URL: *https://doi.org/10.3233/FAIA201017*

[21] Ben-Amram, A. M., Doménech, J. J. and Genaim, S. [2019]. Multiphase-linear ranking functions and their relation to recurrent sets, *in* B. E. Chang (ed.), *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, Vol. 11822 of *Lecture Notes in Computer Science*, Springer, pp. 459–480.
URL: *https://doi.org/10.1007/978-3-030-32304-2_22*

[22] Ben-Amram, A. M. and Genaim, S. [2014]. Ranking functions for linear-constraint loops, *J. ACM* **61**(4): 26:1–26:55.
URL: *https://doi.org/10.1145/2629488*

[23] Ben-Amram, A. M. and Genaim, S. [2017]. On multiphase-linear ranking functions, *in* R. Majumdar and V. Kuncak (eds), *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, Vol. 10427 of *Lecture Notes in Computer Science*, Springer, pp. 601–620.
URL: *https://doi.org/10.1007/978-3-319-63390-9_32*

[24] Beyer, D. [2013]. Second competition on software verification - (summary of SV-COMP 2013), *in* N. Piterman and S. A. Smolka (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013*, Vol. 7795 of *Lecture Notes in Computer Science*, Springer, pp. 594–609.
URL: *https://doi.org/10.1007/978-3-642-36742-7_43*

[25] Beyer, D. [2023]. Competition on software verification and witness validation: SV-COMP 2023, *in* S. Sankaranarayanan and N. Sharygina (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023*, Vol. 13994 of *Lecture Notes in Computer Science*, Springer, pp. 495–522.
URL: *https://doi.org/10.1007/978-3-031-30820-8_29*

[26] Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E. and Sebastiani, R. [2009]. Software model checking via large-block encoding, *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, IEEE, pp. 25–32.
URL: *https://doi.org/10.1109/FMCAD.2009.5351147*

[27] Beyer, D., Jankola, M. and Lingsch-Rosenfeld, M. [2026]. Transition invariants revisited: Termination witnesses and their validation, *Proc. CAV*, Springer.

URL: *https://www.sosy-lab.org/research/pub/2026-CAV.Transition_Invariants_Revisited_Termination_Witnesses_and_Their_Validation.pdf*

[28] Beyer, D., Kettl, M. and Lemberger, T. [2024]. Decomposing software verification using distributed summary synthesis, *Proceedings of the ACM on Software Engineering* **1**(FSE): 1307–1329.

[29] Beyer, D., Lee, N. and Wendler, P. [2022]. Interpolation and sat-based model checking revisited: Adoption to software verification, *CoRR* **abs/2208.05046**.
URL: *https://doi.org/10.48550/arXiv.2208.05046*

[30] Biere, A., Artho, C. and Schuppan, V. [2002]. Liveness checking as safety checking, *in* R. Cleaveland and H. Garavel (eds), *7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems, FMICS 2002, ICALP 2002 Satellite Workshop, Málaga, Spain, July 12-13, 2002*, number 2 in *Electronic Notes in Theoretical Computer Science*, Elsevier, pp. 160–177.
URL: *https://doi.org/10.1016/S1571-0661(04)80410-9*

[31] Biere, A., Cimatti, A., Clarke, E. M., Fujita, M. and Zhu, Y. [1999]. Symbolic model checking using SAT procedures instead of bdds, *in* M. J. Irwin (ed.), *Proceedings of the 36th Conference on Design Automation, New Orleans, LA, USA, June 21-25, 1999*, ACM Press, pp. 317–320.
URL: *https://doi.org/10.1145/309847.309942*

[32] Biere, A., Heule, M., van Maaren, H. and Walsh, T. (eds) [2020]. *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press.
URL: *https://doi.org/10.3233/FAIA336*

[33] Blicha, M., Britikov, K. and Sharygina, N. [2023]. The Golem Horn Solver, under consideration for Computer Aided Verification: 35th International Conference, *CAV 2023, Paris, France, July 15-19, 2023.*, Springer.

[34] Blicha, M., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2022a]. Split transition power abstraction for unbounded safety, *in* A. Griggio and N. Rungta (eds), *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy*, IEEE, pp. 349–358.
URL: *https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_42*

[35] Blicha, M., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2022b]. Transition power abstractions for deep counterexample detection, *in* D. Fisman and G. Rosu (eds), *Tools and Algorithms for the Construction and Analysis of Systems -*

28th International Conference, TACAS 2022, Vol. 13243 of *Lecture Notes in Computer Science*, Springer, pp. 524–542.
URL: *https://doi.org/10.1007/978-3-030-99524-9_29*

[36] Blicha, M., Hyvärinen, A. E. J., Kofron, J. and Sharygina, N. [2019]. Decomposing farkas interpolants, *in* T. Vojnar and L. Zhang (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, Vol. 11427 of *Lecture Notes in Computer Science*, Springer, pp. 3–20.
URL: *https://doi.org/10.1007/978-3-030-17462-0_1*

[37] Blicha, M., Kofron, J. and Tatarko, W. [2022]. Summarization of branching loops, *in* J. Hong, M. Bures, J. W. Park and T. Cerný (eds), *SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing, Virtual Event, April 25 - 29, 2022*, ACM, pp. 1808–1816.
URL: *https://doi.org/10.1145/3477314.3507042*

[38] Böhm, M. and Speckenmeyer, E. [1996]. A fast parallel SAT-solver - efficient workload balancing, *Ann. Math. Artif. Intell. 17* pp. 381–400.

[39] Bombarda, A., Gargantini, A. and Calvagna, A. [2023]. Multi-thread combinatorial test generation with SMT solvers, *in* J. Hong, M. Lanperne, J. W. Park, T. Cerný and H. Shahriar (eds), *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, SAC 2023, Tallinn, Estonia, March 27-31, 2023*, ACM, pp. 1698–1705.
URL: *https://doi.org/10.1145/3555776.3577703*

[40] Bozga, M., Iosif, R. and Konecný, F. [2010]. Fast acceleration of ultimately periodic relations, *in* T. Touili, B. Cook and P. B. Jackson (eds), *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19*, Vol. 6174 of *Lecture Notes in Computer Science*, Springer, pp. 227–242.
URL: *https://doi.org/10.1007/978-3-642-14295-6_23*

[41] Bradley, A. R. [2011]. Sat-based model checking without unrolling, *in* R. Jhala and D. A. Schmidt (eds), *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, Vol. 6538 of *Lecture Notes in Computer Science*, Springer, pp. 70–87.
URL: *https://doi.org/10.1007/978-3-642-18275-4_7*

[42] Bradley, A. R. [2012]. Understanding IC3, *in* A. Cimatti and R. Sebastiani (eds), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, Springer, pp. 1–14.
URL: *https://doi.org/10.1007/978-3-642-31612-8_1*

[43] Bradley, A. R. and Manna, Z. [2008]. Property-directed incremental invariant generation, *Formal Aspects Comput.* **20**(4-5): 379–405.
URL: *https://doi.org/10.1007/s00165-008-0080-9*

[44] Bradley, A. R., Manna, Z. and Sipma, H. B. [2005]. Termination analysis of integer linear loops, *in* M. Abadi and L. de Alfaro (eds), *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, Vol. 3653 of *Lecture Notes in Computer Science*, Springer, pp. 488–502.
URL: *https://doi.org/10.1007/11539452_37*

[45] Braverman, M. [2006]. Termination of integer linear programs, *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 372–385.
URL: *https://doi.org/10.1007/11817963_34*

[46] Brillout, A., Kroening, D., Rümmer, P. and Wahl, T. [2011]. An interpolating sequent calculus for quantifier-free presburger arithmetic, *J. Autom. Reason.* **47**(4): 341–367.
URL: *https://doi.org/10.1007/s10817-011-9237-y*

[47] Britikov, K., Blicha, M., Sharygina, N. and Fedyukovich, G. [2024]. Reachability analysis for multiloop programs using transition power abstraction, *in* A. Platzer, K. Y. Rozier, M. Pradella and M. Rossi (eds), *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*, Vol. 14933 of *Lecture Notes in Computer Science*, Springer, pp. 558–576.
URL: *https://doi.org/10.1007/978-3-031-71162-6_29*

[48] Britikov, K., Fedyukovich, G. and Sharygina, N. [upcoming]. CHC-Based reachability analysis via cycle summarization, *Integrated Formal Methods, iFM 2025, Paris, France*.

[49] Britikov, K., Hyvärinen, A. E. J. and Sharygina, N. [2023]. Picky CDCL: smt-solving with flexible literal selection, *in* A. Reynolds and S. Tasiran (eds), *Verified*

Software. Theories, Tools and Experiments - 15th International Conference, VSTTE 2023, Ames, IA, USA, October 23-24, 2023, Revised Selected Papers, Vol. 14095 of *Lecture Notes in Computer Science*, Springer, pp. 1–19.
URL: *https://doi.org/10.1007/978-3-031-66064-1_1*

[50] Britikov, K., Zlatkin, I., Fedyukovich, G., Alt, L. and Sharygina, N. [2024]. Soltg: A chc-based solidity test case generator, *in* A. Gurfinkel and V. Ganesh (eds), *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, Vol. 14681 of *Lecture Notes in Computer Science*, Springer, pp. 466–479.
URL: *https://doi.org/10.1007/978-3-031-65627-9_23*

[51] Brockschmidt, M., Cook, B., Ishtiaq, S., Khlaaf, H. and Piterman, N. [2016]. T2: temporal property verification, *in* M. Chechik and J. Raskin (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 387–393.
URL: *https://doi.org/10.1007/978-3-662-49674-9_22*

[52] Brockschmidt, M., Ströder, T., Otto, C. and Giesl, J. [2011]. Automated detection of non-termination and nullpointerexceptions for java bytecode, *in* B. Beckert, F. Damiani and D. Gurov (eds), *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2011, Turin, Italy, October 5-7, 2011, Revised Selected Papers*, Lecture Notes in Computer Science, Springer, pp. 123–141.
URL: *https://doi.org/10.1007/978-3-642-31762-0_9*

[53] Bruttomesso, R., Pek, E., Sharygina, N. and Tsitovich, A. [2010]. The opensmt solver, *in* J. Esparza and R. Majumdar (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Vol. 6015 of *Lecture Notes in Computer Science*, Springer, pp. 150–153.
URL: *https://doi.org/10.1007/978-3-642-12002-2_12*

[54] Bryant, R. E. [1986]. Graph-based algorithms for boolean function manipulation, *IEEE Trans. Computers* **35**(8): 677–691.
URL: *https://doi.org/10.1109/TC.1986.1676819*

[55] Bueno, D. [2020]. Horn2vmt: Translating horn reachability into transition

systems., *Technical report*, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[56] Cadar, C., Dunbar, D. and Engler, D. R. [2008]. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs, *in* R. Draves and R. van Renesse (eds), *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, USENIX Association, pp. 209–224.
URL: *http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf*

[57] Cadar, C. and Nowack, M. [2021]. KLEE symbolic execution engine in 2019, *Int. J. Softw. Tools Technol. Transf.* pp. 867–870.
URL: *https://doi.org/10.1007/s10009-020-00570-3*

[58] Chalupa, M., Novák, J. and Strejcek, J. [2021]. Symbiotic 8: Parallel and targeted test generation - (competition contribution), *in* E. Guerra and M. Stoelinga (eds), *Fundamental Approaches to Software Engineering - 24th International Conference, FASE 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, Vol. 12649 of *Lecture Notes in Computer Science*, Springer, pp. 368–372.
URL: *https://doi.org/10.1007/978-3-030-71500-7_20*

[59] Chen, H. Y., Cook, B., Fuhs, C., Nimkar, K. and O'Hearn, P. W. [2014]. Proving nontermination via safety, *in* E. Ábrahám and K. Havelund (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, Vol. 8413 of *Lecture Notes in Computer Science*, Springer, pp. 156–171.
URL: *https://doi.org/10.1007/978-3-642-54862-8_11*

[60] Christ, J., Hoenicke, J. and Nutz, A. [2012]. Smtinterpol: An interpolating SMT solver, *in* A. F. Donaldson and D. Parker (eds), *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, Vol. 7385 of *Lecture Notes in Computer Science*, Springer, pp. 248–254.
URL: *https://doi.org/10.1007/978-3-642-31759-0_19*

[61] Cimatti, A., Griggio, A., Schaafsma, B. J. and Sebastiani, R. [2013]. The mathsat5 SMT solver, *in* N. Piterman and S. A. Smolka (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings, Vol. 7795 of *Lecture Notes in Computer Science*, Springer, pp. 93–107.
URL: *https://doi.org/10.1007/978-3-642-36742-7_7*

[62] Cimatti, A., Griggio, A. and Sebastiani, R. [2008]. Efficient generation of craig interpolants in satisfiability modulo theories, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29 - April 6, 2008. Proceedings*, Vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 397–412.

[63] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. and Veith, H. [2000]. Counterexample-guided abstraction refinement, *in* E. A. Emerson and A. P. Sistla (eds), *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, Vol. 1855 of *Lecture Notes in Computer Science*, Springer, pp. 154–169.
URL: *https://doi.org/10.1007/10722167_15*

[64] Clarke, E. M., Kroening, D., Sharygina, N. and Yorav, K. [2005]. SATABS: sat-based predicate abstraction for ANSI-C, *in* N. Halbwachs and L. D. Zuck (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, Vol. 3440 of *Lecture Notes in Computer Science*, Springer, pp. 570–574.
URL: *https://doi.org/10.1007/978-3-540-31980-1_40*

[65] Cook, B., Podelski, A. and Rybalchenko, A. [2005]. Abstraction refinement for termination, *in* C. Hankin and I. Siveroni (eds), *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 87–101.
URL: *https://doi.org/10.1007/11547662_8*

[66] Cook, B., Podelski, A. and Rybalchenko, A. [2006]. Termination proofs for systems code, *in* M. I. Schwartzbach and T. Ball (eds), *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, ACM, pp. 415–426.
URL: *https://doi.org/10.1145/1133981.1134029*

[67] Craig, W. [1957]. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory, *The Journal of Symbolic Logic* **22**(3): 269–285.

[68] Davis, M., Logemann, G. and Loveland, D. W. [1962]. A machine program for theorem-proving, *Commun. ACM 5* pp. 394–397.

[69] De Angelis, E. and K., H. G. V. [2022]. CHC-COMP 2022: Competition report, *in* G. W. Hamilton, T. Kahsai and M. Proietti (eds), *Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022*, Vol. 373 of *EPTCS*, pp. 44–62.

[70] De Angelis, E. and K., H. G. V. [2023]. CHC-COMP 2023: Competition report, *in* T. Kutsia, D. Ventura, D. Monniaux and J. F. Morales (eds), *Proceedings 18th International Workshop on Logical and Semantic Frameworks, with Applications and 10th Workshop on Horn Clauses for Verification and Synthesis, LSFA/HCVS 2023, and 10th Workshop on Horn Clauses for Verification and SynthesisRome, Italy & Paris, France, 1-2 July, 2023 & 23rd April 2023*, Vol. 402 of *EPTCS*, pp. 83–104.
URL: *https://doi.org/10.4204/EPTCS.402.10*

[71] De Angelis, E. and Krishnan, H. G. V. [2023]. Competition of solvers for constrained horn clauses (CHC-COMP 2023), *in* D. Beyer, A. Hartmanns and F. Kordon (eds), *TOOLympics Challenge 2023 - Updates, Results, Successes of the Formal-Methods Competitions, TOOLympics 2023, Paris, France*, Vol. 14550 of *Lecture Notes in Computer Science*, Springer, pp. 38–51.
URL: *https://doi.org/10.1007/978-3-031-67695-6_2*

[72] de Moura, L. M. and Bjørner, N. S. [2008]. Z3: an efficient SMT solver, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008*, Vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.
URL: *https://doi.org/10.1007/978-3-540-78800-3_24*

[73] Dijkstra, E. W. [1976]. Formal techniques and sizeable programs, *in* K. Samelson (ed.), *ECI Conference 1976, Proceedings of the 1st European Cooperation in Informatics, Amsterdam, The Netherlands, August 9-12, 1976, Proceedings*, Vol. 44 of *Lecture Notes in Computer Science*, Springer, pp. 225–235.
URL: *https://doi.org/10.1007/3-540-07804-5_31*

[74] Donaldson, A. F., Haller, L., Kroening, D. and Rümmer, P. [2011]. Software verification using k-induction, *in* E. Yahav (ed.), *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*,

Vol. 6887 of *Lecture Notes in Computer Science*, Springer, pp. 351–368.
URL: *https://doi.org/10.1007/978-3-642-23702-7_26*

[75] Donaldson, A. F., Kroening, D. and Rümmer, P. [2011]. Automatic analysis of DMA races using model checking and *k*-induction, *Formal Methods Syst. Des.* **39**(1): 83–113.
URL: *https://doi.org/10.1007/s10703-011-0124-2*

[76] D'Silva, V. [2010]. Propositional interpolation and abstract interpretation, *in* A. D. Gordon (ed.), *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, Vol. 6012 of *Lecture Notes in Computer Science*, Springer, pp. 185–204.

[77] D'Silva, V., Kroening, D., Purandare, M. and Weissenbacher, G. [2010]. Interpolant strength, *in* G. Barthe and M. V. Hermenegildo (eds), *Verification, Model Checking, and Abstract Interpretation, 11th International Conference, VMCAI 2010, Madrid, Spain, January 17-19, 2010. Proceedings*, Vol. 5944 of *Lecture Notes in Computer Science*, Springer, pp. 129–145.

[78] Dutertre, B. [2014]. Yices 2.2, *in* A. Biere and R. Bloem (eds), *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, Vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.
URL: *https://doi.org/10.1007/978-3-319-08867-9_49*

[79] Eén, N. and Sörensson, N. [2003]. An extensible sat-solver, *in* E. Giunchiglia and A. Tacchella (eds), *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers*, Vol. 2919 of *Lecture Notes in Computer Science*, Springer, pp. 502–518.
URL: *https://doi.org/10.1007/978-3-540-24605-3_37*

[80] Emerson, E. A. and Clarke, E. M. [1982]. Using branching time temporal logic to synthesize synchronization skeletons, *Sci. Comput. Program.* **2**(3): 241–266.
URL: *https://doi.org/10.1016/0167-6423(83)90017-5*

[81] Ernst, G. [2023]. Korn - software verification with horn clauses (competition contribution), *in* S. Sankaranarayanan and N. Sharygina (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference,*

TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II, Vol. 13994 of *Lecture Notes in Computer Science*, Springer, pp. 559–564.
URL: *https://doi.org/10.1007/978-3-031-30820-8_36*

[82] Esen, Z. and Rümmer, P. [2022]. Tricera: Verifying C programs using the theory of heaps, *in* A. Griggio and N. Rungta (eds), *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, IEEE, pp. 380–391.
URL: *https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_45*

[83] Fedyukovich, G., Kaufman, S. J. and Bodík, R. [2020]. Learning inductive invariants by sampling from frequency distributions, *Formal Methods Syst. Des.* **56**(1): 154–177.
URL: *https://doi.org/10.1007/s10703-020-00349-x*

[84] Fedyukovich, G., Prabhu, S., Madhukar, K. and Gupta, A. [2019]. Quantified invariants via syntax-guided synthesis, *in* I. Dillig and S. Tasiran (eds), *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, Vol. 11561 of *Lecture Notes in Computer Science*, Springer, pp. 259–277.
URL: *https://doi.org/10.1007/978-3-030-25540-4_14*

[85] Fedyukovich, G., Zhang, Y. and Gupta, A. [2018]. Syntax-guided termination analysis, *in* H. Chockler and G. Weissenbacher (eds), *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*, Lecture Notes in Computer Science, Springer, pp. 124–143.
URL: *https://doi.org/10.1007/978-3-319-96145-3_7*

[86] Feist, J., Grieco, G. and Groce, A. [2019]. Slither: a static analysis framework for smart contracts, *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, IEEE / ACM, pp. 8–15.
URL: *https://doi.org/10.1109/WETSEB.2019.00008*

[87] Floyd, R. W. [1993]. Assigning meanings to programs, *Program Verification: Fundamental Issues in Computer Science*, Springer, pp. 65–81.

[88] Frohn, F. [2020]. A calculus for modular loop acceleration, *in* A. Biere and D. Parker (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020*, Vol. 12078 of *Lecture Notes in*

Computer Science, Springer, pp. 58–76.
URL: *https://doi.org/10.1007/978-3-030-45190-5_4*

[89] Frohn, F. and Giesl, J. [2019]. Proving non-termination via loop acceleration, *CoRR* **abs/1905.11187**.
URL: *http://arxiv.org/abs/1905.11187*

[90] Frohn, F. and Giesl, J. [2022]. Proving non-termination and lower runtime bounds with loat (system description), *in* J. Blanchette, L. Kovács and D. Pattinson (eds), *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, Vol. 13385 of *Lecture Notes in Computer Science*, Springer, pp. 712–722.
URL: *https://doi.org/10.1007/978-3-031-10769-6_41*

[91] Frohn, F. and Giesl, J. [2024a]. Accelerated bounded model checking, *CoRR* **abs/2401.09973**.
URL: *https://doi.org/10.48550/arXiv.2401.09973*

[92] Frohn, F. and Giesl, J. [2024b]. Integrating loop acceleration into bounded model checking, *in* A. Platzer, K. Y. Rozier, M. Pradella and M. Rossi (eds), *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, September 9-13, 2024, Proceedings, Part I*, Vol. 14933 of *Lecture Notes in Computer Science*, Springer, pp. 73–91.
URL: *https://doi.org/10.1007/978-3-031-71162-6_4*

[93] Frohn, F. and Giesl, J. [2025]. Infinite state model checking by learning transitive relations, *in* C. W. Barrett and U. Waldmann (eds), *Automated Deduction - CADE 30 - 30th International Conference on Automated Deduction, Stuttgart, Germany, July 28-31, 2025, Proceedings*, Vol. 15943 of *Lecture Notes in Computer Science*, Springer, pp. 468–489.
URL: *https://doi.org/10.1007/978-3-031-99984-0_25*

[94] Frohn, F., Hark, M. and Giesl, J. [2020]. Termination of polynomial loops, *in* D. Pichardie and M. Sighireanu (eds), *Static Analysis - 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 89–112.
URL: *https://doi.org/10.1007/978-3-030-65474-0_5*

[95] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A. and Tinelli, C. [2004]. DPLL(T): fast decision procedures, *in* R. Alur and D. A. Peled (eds), *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July*

13-17, 2004, Proceedings, Vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 175–188.
URL: *https://doi.org/10.1007/978-3-540-27813-9_14*

[96] Giesl, J., Aschermann, C., Brockschmidt, M., Emmes, F., Frohn, F., Fuhs, C., Hensel, J., Otto, C., Plücker, M., Schneider-Kamp, P., Ströder, T., Swiderski, S. and Thiemann, R. [2017]. Analyzing program termination and complexity automatically with aprove, *J. Autom. Reason.* **58**(1): 3–31.
URL: *https://doi.org/10.1007/s10817-016-9388-y*

[97] Giesl, J., Mesnard, F., Rubio, A., Thiemann, R. and Waldmann, J. [2015]. Termination competition (termcomp 2015), *in* A. P. Felty and A. Middeldorp (eds), *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 105–108.
URL: *https://doi.org/10.1007/978-3-319-21401-6_6*

[98] Giesl, J., Rubio, A., Sternagel, C., Waldmann, J. and Yamada, A. [2019]. The termination and complexity competition, *in* D. Beyer, M. Huisman, F. Kordon and B. Steffen (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 25 Years of TACAS: TOOLympics, Held as Part of ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part III*, Vol. 11429 of *Lecture Notes in Computer Science*, Springer, pp. 156–166.
URL: *https://doi.org/10.1007/978-3-030-17502-3_10*

[99] Godefroid, P. [2020]. Fuzzing: hack, art, and science, *Commun. ACM* **63**(2): 70–76.
URL: *https://doi.org/10.1145/3363824*

[100] Godefroid, P., Levin, M. Y. and Molnar, D. A. [2008]. Automated whitebox fuzz testing, *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*, The Internet Society.
URL: *https://www.ndss-symposium.org/ndss2008/automated-whitebox-fuzz-testing/*

[101] Godefroid, P. and Luchaup, D. [2011]. Automatic partial loop summarization in dynamic test generation, *in* M. B. Dwyer and F. Tip (eds), *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada*, ACM, pp. 23–33.
URL: *https://doi.org/10.1145/2001420.2001424*

[102] Graf, S. and Saïdi, H. [1997]. Construction of abstract state graphs with PVS, *in* O. Grumberg (ed.), *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, Vol. 1254 of *Lecture Notes in Computer Science*, Springer, pp. 72–83.
URL: *https://doi.org/10.1007/3-540-63166-6_10*

[103] Grebenshchikov, S., Lopes, N. P., Popeea, C. and Rybalchenko, A. [2012]. Synthesizing software verifiers from proof rules, *in* J. Vitek, H. Lin and F. Tip (eds), *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, ACM, pp. 405–416.
URL: *https://doi.org/10.1145/2254064.2254112*

[104] Grieco, G., Song, W., Cygan, A., Feist, J. and Groce, A. [2020]. Echidna: Effective, usable, and fast fuzzing for smart contracts, *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2020*, ACM, pp. 557–560.

[105] Gupta, A., Henzinger, T. A., Majumdar, R., Rybalchenko, A. and Xu, R. [2008]. Proving non-termination, *in* G. C. Necula and P. Wadler (eds), *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, ACM, pp. 147–158.
URL: *https://doi.org/10.1145/1328438.1328459*

[106] Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. [2015]. The seahorn verification framework, *in* D. Kroening and C. S. Pasareanu (eds), *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, Vol. 9206 of *Lecture Notes in Computer Science*, Springer, pp. 343–361.
URL: *https://doi.org/10.1007/978-3-319-21690-4_20*

[107] Hajdu, Á. and Jovanović, D. [2020]. solc-verify: A modular verifier for Solidity smart contracts, *Verified Software. Theories, Tools, and Experiments, VSTTE 2019*, Lecture Notes in Computer Science, Springer, pp. 161–179.

[108] Hark, M., Frohn, F. and Giesl, J. [2025]. Termination of triangular polynomial loops, *Formal Methods Syst. Des.* **65**(1): 70–132.
URL: *https://doi.org/10.1007/s10703-023-00440-z*

[109] Heizmann, M., Hoenicke, J. and Podelski, A. [2014]. Termination analysis by learning terminating programs, *in* A. Biere and R. Bloem (eds), *Computer Aided*

Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, Lecture Notes in Computer Science, Springer, pp. 797–813.
URL: *https://doi.org/10.1007/978-3-319-08867-9_53*

[110] Henzinger, T. A., Jhala, R., Majumdar, R. and Sutre, G. [2002]. Lazy abstraction, *in* J. Launchbury and J. C. Mitchell (eds), *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, ACM, pp. 58–70.
URL: *https://doi.org/10.1145/503272.503279*

[111] Heule, M. J. H. [2016]. The DRAT format and drat-trim checker, *CoRR abs/1610.06229* .

[112] Heule, M. J. H. and van Maaren, H. [2021]. Look-ahead based SAT solvers, *in* A. Biere, M. Heule, H. van Maaren and T. Walsh (eds), *Handbook of Satisfiability - Second Edition*, Vol. 336 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, pp. 183–212.

[113] Heule, M., Kullmann, O., Wieringa, S. and Biere, A. [2011]. Cube and conquer: Guiding CDCL SAT solvers by lookaheads, *in* K. Eder, J. Lourencco and O. Shehory (eds), *Hardware and Software: Verification and Testing - 7th International Haifa Verification Conference, HVC 2011, Haifa, Israel, December 6-8, 2011, Revised Selected Papers*, Vol. 7261 of *Lecture Notes in Computer Science*, Springer, pp. 50–65.

[114] Hoare, C. A. R. [1969]. An axiomatic basis for computer programming, *Communications of the ACM* **12**(10): 576–580.

[115] Hoder, K. and Bjørner, N. S. [2012]. Generalized property directed reachability, *in* A. Cimatti and R. Sebastiani (eds), *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, Vol. 7317 of *Lecture Notes in Computer Science*, Springer, pp. 157–171.
URL: *https://doi.org/10.1007/978-3-642-31612-8_13*

[116] Hojjat, H. and Rümmer, P. [2018]. The ELDARICA horn solver, *in* N. S. Bjørner and A. Gurfinkel (eds), *Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA*, IEEE, pp. 1–7.
URL: *https://doi.org/10.23919/FMCAD.2018.8603013*

[117] Hyvärinen, A. E. J., Marescotti, M., Alt, L. and Sharygina, N. [2016]. Opensmt2: An SMT solver for multi-core and cloud computing, *in* N. Creignou and D. L. Berre (eds), *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, Vol. 9710 of *Lecture Notes in Computer Science*, Springer, pp. 547–553.
URL: *https://doi.org/10.1007/978-3-319-40970-2_35*

[118] Hyvärinen, A. E. J., Marescotti, M., Sadigova, P., Chockler, H. and Sharygina, N. [2018]. Lookahead-based SMT solving, *in* G. Barthe, G. Sutcliffe and M. Veanes (eds), *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*, Vol. 57 of *EPiC Series in Computing*, EasyChair, pp. 418–434.

[119] Jovanovic, D. and Dutertre, B. [2016]. Property-directed k-induction, *in* R. Piskac and M. Talupur (eds), *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, IEEE, pp. 85–92.
URL: *https://doi.org/10.1109/FMCAD.2016.7886665*

[120] Kafle, B., Gallagher, J. P. and Morales, J. F. [2016]. Rahft: A tool for verifying horn clauses using abstract interpretation and finite tree automata, *in* S. Chaudhuri and A. Farzan (eds), *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada*, Vol. 9779 of *Lecture Notes in Computer Science*, Springer, pp. 261–268.
URL: *https://doi.org/10.1007/978-3-319-41528-4_14*

[121] Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S. K., Lal, A., Rastogi, A., Roy, S. and Sharma, R. [2023]. Finding inductive loop invariants using large language models, *CoRR* **abs/2311.07948**.
URL: *https://doi.org/10.48550/arXiv.2311.07948*

[122] Kapur, D., Majumdar, R. and Zarba, C. G. [2006]. Interpolation for data structures, *in* M. Young and P. T. Devanbu (eds), *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, ACM, pp. 105–116.
URL: *https://doi.org/10.1145/1181775.1181789*

[123] King, J. C. [1976]. Symbolic execution and program testing, *Commun. ACM* **19**(7): 385–394.
URL: *https://doi.org/10.1145/360248.360252*

[124] Komuravelli, A., Gurfinkel, A. and Chaki, S. [2016]. Smt-based model checking for recursive programs, *Formal Methods Syst. Des.* **48**(3): 175–205.
URL: *https://doi.org/10.1007/s10703-016-0249-4*

[125] Konnov, I. [2019]. Edmund m. clarke, thomas a. henzinger, helmut veith, and roderick bloem (eds): Handbook of model checking - springer international publishing ag, cham, switzerland, 2018, *Formal Aspects Comput.* **31**(4): 455–456.
URL: *https://doi.org/10.1007/s00165-019-00486-z*

[126] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A. and Wintersteiger, C. M. [2008]. Loop summarization using abstract transformers, *in* S. D. Cha, J. Choi, M. Kim, I. Lee and M. Viswanathan (eds), *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea*, Vol. 5311 of *Lecture Notes in Computer Science*, Springer, pp. 111–125.
URL: *https://doi.org/10.1007/978-3-540-88387-6_10*

[127] Kroening, D., Sharygina, N., Tsitovich, A. and Wintersteiger, C. M. [2010]. Termination analysis with compositional transition invariants, *in* T. Touili, B. Cook and P. B. Jackson (eds), *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 89–103.
URL: *https://doi.org/10.1007/978-3-642-14295-6_9*

[128] Kura, S., Unno, H. and Hasuo, I. [2021]. Decision tree learning in cegis-based termination analysis, *in* A. Silva and K. R. M. Leino (eds), *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*, Lecture Notes in Computer Science, Springer, pp. 75–98.
URL: *https://doi.org/10.1007/978-3-030-81688-9_4*

[129] Labbaf, F., Kolárik, T., Blicha, M., Fedyukovich, G., Wand, M. and Sharygina, N. [2025]. Space explanations of neural network classification, *CoRR* **abs/2511.22498**.
URL: *https://doi.org/10.48550/arXiv.2511.22498*

[130] Lahiri, S. K., Nieuwenhuis, R. and Oliveras, A. [2006]. SMT techniques for fast predicate abstraction, *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, Vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 424–437.
URL: *https://doi.org/10.1007/11817963_39*

[131] Lamport, L. [1977]. Proving the correctness of multiprocess programs, *IEEE Trans. Software Eng.* **3**(2): 125–143.
URL: *https://doi.org/10.1109/TSE.1977.229904*

[132] Larraz, D., Nimkar, K., Oliveras, A., Rodríguez-Carbonell, E. and Rubio, A. [2014]. Proving non-termination using max-smt, *in* A. Biere and R. Bloem (eds), *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 779–796.
URL: *https://doi.org/10.1007/978-3-319-08867-9_52*

[133] Le, T. C., Qin, S. and Chin, W. [2015]. Termination and non-termination specification inference, *in* D. Grove and S. M. Blackburn (eds), *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, ACM, pp. 489–498.
URL: *https://doi.org/10.1145/2737924.2737993*

[134] Leike, J. and Heizmann, M. [2018]. Geometric nontermination arguments, *in* D. Beyer and M. Huisman (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, Lecture Notes in Computer Science, Springer, pp. 266–283.
URL: *https://doi.org/10.1007/978-3-319-89963-3_16*

[135] Leopardi, F., Labbaf, F., Kolárik, T., Wand, M. and Sharygina, N. [2026]. Interpreting logical explanations of classifying neural networks, *34th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2026, Bruges, Belgium, April 22-24, 2026*.
URL: *https://doi.org/10.14428/esann/2026.ES2026-326*

[136] Lin, S., Sun, J., Xiao, H., Liu, Y., Sanán, D. and Hansen, H. [2017]. Fib: squeezing loop invariants by interpolation between forward/backward predicate transformers, *in* G. Rosu, M. D. Penta and T. N. Nguyen (eds), *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA*, IEEE Computer Society, pp. 793–803.
URL: *https://doi.org/10.1109/ASE.2017.8115690*

[137] Lommen, N. and Giesl, J. [2025]. Aprove (koat + loat), *in* A. Gurfinkel and M. Heule (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Nature Switzerland, Cham, pp. 205–211.

[138] Luu, L., Chu, D., Olickel, H., Saxena, P. and Hobor, A. [2016]. Making smart contracts smarter, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS 2016*, ACM, pp. 254–269.

[139] Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A. E. J. and Sharygina, N. [2020]. Accurate smart contract verification through direct modelling, *in* T. Margaria and B. Steffen (eds), *Leveraging Applications of Formal Methods, Verification and Validation: Applications - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part III*, Vol. 12478 of *Lecture Notes in Computer Science*, Springer, pp. 178–194.
URL: *https://doi.org/10.1007/978-3-030-61467-6_12*

[140] Matsushita, Y., Tsukada, T. and Kobayashi, N. [2020]. Rusthorn: Chc-based verification for rust programs (full version), *CoRR* **abs/2002.09002**.
URL: *https://arxiv.org/abs/2002.09002*

[141] 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.
URL: *https://doi.org/10.1145/3462205*

[142] McMillan, K. L. [2003]. Interpolation and sat-based model checking, *in* W. A. H. Jr. and F. Somenzi (eds), *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, Vol. 2725 of *Lecture Notes in Computer Science*, Springer, pp. 1–13.
URL: *https://doi.org/10.1007/978-3-540-45069-6_1*

[143] McMillan, K. L. [2004]. An interpolating theorem prover, *in* K. Jensen and A. Podelski (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, Vol. 2988 of *Lecture Notes in Computer Science*, Springer, pp. 16–30.
URL: *https://doi.org/10.1007/978-3-540-24730-2_2*

[144] McMillan, K. L. [2005]. Applications of craig interpolants in model checking, *in* N. Halbwachs and L. D. Zuck (eds), *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, Vol. 3440 of *Lecture Notes in Computer Science*, Springer, pp. 1–12.

[145] McMillan, K. L. [2006]. Lazy abstraction with interpolants, *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, Vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 123–136.

[146] McMillan, K. L. [2010]. Lazy annotation for program testing and verification, *Computer Aided Verification, CAV 2010*, Lecture Notes in Computer Science, Springer, pp. 104–118.

[147] McMillan, K. L. [2014]. Lazy annotation revisited, *Computer Aided Verification, CAV 2014*, Lecture Notes in Computer Science, Springer, pp. 243–259.

[148] McMillan, K. and Rybalchenko, A. [2013]. Computing relational fixed points using interpolation, *Technical report, January 2013. MSR-TR-2013-6* .

[149] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L. and Malik, S. [2001]. Chaff: Engineering an efficient SAT solver, *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, ACM, pp. 530–535.
URL: *https://doi.org/10.1145/378239.379017*

[150] Nielson, H. R. and Nielson, F. [1992]. *Semantics with applications - a formal introduction*, Wiley professional computing, Wiley.

[151] Oh, C. [2015]. Between SAT and UNSAT: the fundamental difference in CDCL SAT, *in* M. Heule and S. A. Weaver (eds), *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, Vol. 9340 of *Lecture Notes in Computer Science*, Springer, pp. 307–323.

[152] Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A. E. J. and Sharygina, N. [2021]. Theory-specific proof steps witnessing correctness of SMT executions, *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, IEEE, pp. 541–546.

[153] Ozdemir, A., Kremer, G., Tinelli, C. and Barrett, C. W. [2023]. Satisfiability modulo finite fields, *in* C. Enea and A. Lal (eds), *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, Vol. 13965 of *Lecture Notes in Computer Science*, Springer, pp. 163–186.
URL: *https://doi.org/10.1007/978-3-031-37703-7_8*

[154] Peña, R., Sánchez-Hernández, J., Garrido, M. and Sagredo, J. [2023]. Smt-based test-case generation and validation for programs with complex specifications, *in* P. López-García, J. P. Gallagher and R. Giacobazzi (eds), *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems - Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*, Vol. 13160 of *Lecture Notes in Computer Science*, Springer, pp. 188–205.
URL: *https://doi.org/10.1007/978-3-031-31476-6_10*

[155] Permenev, A., Dimitrov, D. K., Tsankov, P., Drachsler-Cohen, D. and Vechev, M. T. [2020]. Verx: Safety verification of smart contracts, *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, pp. 1661–1677.
URL: *https://doi.org/10.1109/SP40000.2020.00024*

[156] Podelski, A. and Rybalchenko, A. [2004a]. A complete method for the synthesis of linear ranking functions, *in* B. Steffen and G. Levi (eds), *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, Vol. 2937 of *Lecture Notes in Computer Science*, Springer, pp. 239–251.
URL: *https://doi.org/10.1007/978-3-540-24622-0_20*

[157] Podelski, A. and Rybalchenko, A. [2004b]. Transition invariants, *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, IEEE Computer Society, pp. 32–41.
URL: *https://doi.org/10.1109/LICS.2004.1319598*

[158] Pudlák, P. [1997]. Lower bounds for resolution and cutting plane proofs and monotone computations, *J. Symb. Log.* **62**(3): 981–998.

[159] Queille, J. and Sifakis, J. [1982]. Specification and verification of concurrent systems in CESAR, *in* M. Dezani-Ciancaglini and U. Montanari (eds), *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings*, Vol. 137 of *Lecture Notes in Computer Science*, Springer, pp. 337–351.
URL: *https://doi.org/10.1007/3-540-11494-7_22*

[160] Riley, D. and Fedyukovich, G. [2025]. Exact loop bound analysis, *Proc. ACM Program. Lang.* **9**(PLDI): 1814–1837.
URL: *https://doi.org/10.1145/3729323*

[161] Rümmer, P. [2019]. Jayhorn: a java model checker, *in* T. Murray and G. Ernst (eds), *Proceedings of the 21st Workshop on Formal Techniques for Java-like Pro-*

grams, FTfJP@ECOOP 2019, London, United Kingdom, July 15, 2019, ACM, p. 1:1.
URL: *https://doi.org/10.1145/3340672.3341113*

[162] Ryan, G., Wong, J., Yao, J., Gu, R. and Jana, S. [2020]. CLN2INV: learning loop invariants with continuous logic networks, *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia*, OpenReview.net.
URL: *https://openreview.net/forum?id=HJlfuTEtvB*

[163] Sarita, Y., Singh, A., Gomber, S., Singh, G. and Vishwanathan, M. [2026]. Efficient ranking function-based termination analysis via bidirectional decompositional search, *in* R. Krebbers (ed.), *Programming Languages and Systems - 35th European Symposium on Programming, ESOP 2026, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part II*, Lecture Notes in Computer Science, Springer, pp. 181–209.
URL: *https://doi.org/10.1007/978-3-032-22723-2_7*

[164] Seshia, S. A., Sharygina, N. and Tripakis, S. [2018]. Modeling for verification, *in* E. M. Clarke, T. A. Henzinger, H. Veith and R. Bloem (eds), *Handbook of Model Checking*, Springer, pp. 75–105.
URL: *https://doi.org/10.1007/978-3-319-10575-8_3*

[165] Sheeran, M., Singh, S. and Stålmarck, G. [2000]. Checking safety properties using induction and a sat-solver, *in* W. A. H. Jr. and S. D. Johnson (eds), *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, Vol. 1954 of *Lecture Notes in Computer Science*, Springer, pp. 108–125.
URL: *https://doi.org/10.1007/3-540-40922-X_8*

[166] Silva, J. P. M. and Sakallah, K. A. [1996a]. Conflict analysis in search algorithms for satisfiability, *Eigth International Conference on Tools with Artificial Intelligence, ICTAI '96, Toulouse, France, November 16-19, 1996*, IEEE Computer Society, pp. 467–469.

[167] Silva, J. P. M. and Sakallah, K. A. [1996b]. GRASP - a new search algorithm for satisfiability, *in* R. A. Rutenbar and R. H. J. M. Otten (eds), *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, IEEE Computer Society / ACM, pp. 220–227.
URL: *https://doi.org/10.1109/ICCAD.1996.569607*

[168] Silverman, J. and Kincaid, Z. [2019]. Loop summarization with rational vector addition systems, *in* I. Dillig and S. Tasiran (eds), *Computer Aided Verification - 31st International Conference, CAV 2019*, Vol. 11562 of *Lecture Notes in Computer Science*, Springer, pp. 97–115.
URL: *https://doi.org/10.1007/978-3-030-25543-5_7*

[169] Strejcek, J. and Trtík, M. [2012]. Abstracting path conditions, *in* M. P. E. Heimdahl and Z. Su (eds), *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA*, ACM, pp. 155–165.
URL: *https://doi.org/10.1145/2338965.2336772*

[170] Tiwari, A. [2004]. Termination of linear programs, *in* R. Alur and D. A. Peled (eds), *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, Vol. 3114 of *Lecture Notes in Computer Science*, Springer, pp. 70–82.
URL: *https://doi.org/10.1007/978-3-540-27813-9_6*

[171] Tsitovich, A., Sharygina, N., Wintersteiger, C. M. and Kroening, D. [2011]. Loop summarization and termination analysis, *in* P. A. Abdulla and K. R. M. Leino (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, Lecture Notes in Computer Science, Springer, pp. 81–95.
URL: *https://doi.org/10.1007/978-3-642-19835-9_9*

[172] Turing, A. [1949]. Checking a large routine, *Report of a Conference on High Speed Automatic Calculating Machines*, pp. pp.67–69.

[173] Unno, H., Terauchi, T., Gu, Y. and Koskinen, E. [2023]. Modular primal-dual fixpoint logic solving for temporal verification, *Proc. ACM Program. Lang.* **7**(POPL): 2111–2140.
URL: *https://doi.org/10.1145/3571265*

[174] Urban, C., Gurfinkel, A. and Kahsai, T. [2016]. Synthesizing ranking functions from bits and pieces, *in* M. Chechik and J. Raskin (eds), *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Lecture Notes in Computer Science, Springer, pp. 54–70.
URL: *https://doi.org/10.1007/978-3-662-49674-9_4*

[175] Vizel, Y., Weissenbacher, G. and Malik, S. [2015]. Boolean satisfiability solvers and their applications in model checking, *Proc. IEEE* **103**(11): 2021–2035.
URL: *https://doi.org/10.1109/JPROC.2015.2455034*

[176] Wang, S., Zhang, H., Xu, K., Lin, X., Jana, S., Hsieh, C. and Kolter, J. Z. [2021]. Beta-crown: Efficient bound propagation with per-neuron split constraints for neural network robustness verification, *in* M. Ranzato, A. Beygelzimer, Y. N. Dauphin, P. Liang and J. W. Vaughan (eds), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 29909–29921.
URL: *https://proceedings.neurips.cc/paper/2021/hash/fac7fead96dafceaf80c1daffeae82a4-Abstract.html*

[177] Wesley, S., Christakis, M., Navas, J. A., Trefler, R. J., Wüstholz, V. and Gurfinkel, A. [2022]. Verifying solidity smart contracts via communication abstraction in smartace, *in* B. Finkbeiner and T. Wies (eds), *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, Vol. 13182 of *Lecture Notes in Computer Science*, Springer, pp. 425–449.
URL: *https://doi.org/10.1007/978-3-030-94583-1_21*

[178] Wetzler, N., Heule, M. and Jr., W. A. H. [2014]. Drat-trim: Efficient checking and trimming using expressive clausal proofs, *in* C. Sinz and U. Egly (eds), *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, Vol. 8561 of *Lecture Notes in Computer Science*, Springer, pp. 422–429.

[179] Wu, M., Wu, H. and Barrett, C. W. [2023]. Verix: Towards verified explainability of deep neural networks, *in* A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt and S. Levine (eds), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.
URL: *http://papers.nips.cc/paper_files/paper/2023/hash/46907c2ff9fafd618095161d76461842-Abstract-Conference.html*

[180] Xiao, F., Li, C., Luo, M., Manyà, F., Lü, Z. and Li, Y. [2019]. A branching heuristic for SAT solvers based on complete implication graphs, *Sci. China Inf. Sci. 62* pp. 72103:1–72103:13.

[181] Xie, X., Chen, B., Zou, L., Liu, Y., Le, W. and Li, X. [2019]. Automatic loop summarization via path dependency analysis, *IEEE Trans. Software Eng.* **45**(6): 537–557.
URL: *https://doi.org/10.1109/TSE.2017.2788018*

[182] Zlatkin, I. and Fedyukovich, G. [2022]. Maximizing branch coverage with constrained horn clauses, *in* D. Fisman and G. Rosu (eds), *TACAS, Part II*, Vol. 13244 of *Lecture Notes in Computer Science*, Springer, pp. 254–272.