FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

DOCTORAL THESIS

Martin Blicha

Effective Automated Software Verification: A Multilayered Approach

Department of Distributed and Dependable Systems

Supervisor of the doctoral thesis:	doc. RNDr. Jan Kofroň, Ph.D.
	Prof. Natasha Sharygina
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2023

Effective Automated Software Verification: A Multilayered Approach

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
Martin Blicha

under the supervision of
Natasha Sharygina and Jan Kofroň

January 2023

Dissertation Committee

Matthias Hauswirth	Università della Svizzera italiana, Switzerland
Patrick Eugster	Università della Svizzera italiana, Switzerland
Pavel Parízek	Charles University, Prague, Czech Republic
Nikolaj Bjørner	Microsoft Research, Redmont, USA
Philipp Rümmer	University of Regensburg, Regensburg, Germany

Dissertation accepted on 25 January 2023

Natasha Sharygina
Research Advisor
Università della Svizzera italiana, Switzerland

Jan Kofroň
Research Advisor
Charles University, Prague, Czech Republic

Walter Binder
PhD Program Director

Martin Blicha
Lugano, 25 January 2023

Abstract

In recent years, automated formal verification of software has progressed from a few research labs into large-scale applications, such as cloud infrastructure and smart contracts. Formal verification techniques based on model checking provide the necessary guarantees by exploring systems' behaviour *exhaustively* and *automatically*. Moreover, they provide *witnesses* (explanations) for the result of their analysis: a faulty behaviour, if there exists one, or a proof of the absence of such behaviour.

However, the general problem that automated software verification is trying to solve is undecidable. Despite this theoretical barrier, it is quite efficient on many instances that arise in practice. We ascribe this (perhaps surprising) success to a combination of factors: the relentless effort of researchers that come up with new verification procedures to tackle classes of problems where existing techniques struggle; amazing progress in the foundational technologies of satisfiability solving, especially in *Satisfiability Modulo Theories* (SMT); and increase of available computational power through parallel and cloud computing. Nevertheless, the growing complexity of real-world systems poses new challenges for formal verification, especially for the scalability of the techniques.

The task of automated software verification has two parts: modelling the task in a formal framework and solving the resulting mathematical problem. While modelling is a non-trivial step in the verification process, it has been addressed widely, and there exist numerous modelling concepts suitable for various systems. Solving, on the other hand, is a bottleneck when it comes to complex modern programs. This thesis focuses on the solving part of the task, where there is a need for new effective solutions. We assume the problems are modelled *symbolically*, with formulas in first-order logic. Specifically, we work in the logical framework of *constrained Horn clauses* (CHC) and research the mathematical problem of *deciding satisfiability* of a CHC system. CHC satisfiability generalizes the common task of verifying *safety properties* in *transition systems*, a widespread model in formal verification. This task is complex and undecidable in general already if the language of the constraints contains linear integer arithmetic. In our work, we argue that this task can be approached by providing solutions at different levels, which we identify as *foundational*, *verification* and *cooperative* layers of the problem. These correspond to decision and interpolation procedures, sequential model-checking algorithms, and multi-agent solving approaches. We further argue that the next (higher) layers build on, and interact with, the previous (lower) layers and that working on the higher layers can significantly benefit from a deep understanding of the layers beneath them. Overall, we advance the field of automated software verification by contributing solutions on all three layers.

On the foundational layer, we contribute a new interpolation algorithm for conflicts in the theory of linear arithmetic. It extends the standard approach based on the Farkas lemma and can compute logically stronger interpolants. Experimental evaluation in a model-checking scenario shows that with our interpolation algorithm, the same model-checking algorithm can successfully solve some problems on which it diverges using the original interpolation algorithm.

On the verification layer, we invent the concept of *transition power abstraction* (TPA) sequence and contribute TPA-based model-checking algorithms that address the known problem of detecting deep counterexamples in transition systems. Moreover, we show that the TPA sequence can be mined for candidates for *transition* invariants. This allows TPA-based algorithms to prove systems safe by means largely orthogonal to existing techniques.

To support the development of verification techniques, we contribute Golem, a new solver for the satisfiability of systems of constrained Horn clauses. The main features of Golem are its tight integration with the underlying interpolating SMT solver and support for multiple back-end solving algorithms. Golem is primarily meant to serve as a research tool for further investigation of SMT-based algorithms for model checking and general Horn solving. It was instrumental in developing our prototype implementation of the TPA-based algorithms. However, it is also efficient compared to other Horn solvers in the latest edition of CHC-COMP. As such, it can be used as the back end for domain-specific tools that model various verification tasks in the CHC framework. It has already been included as a possible back end for the software verifier Korn.

On the cooperative layer, we contribute an abstract framework that generalizes concepts from induction-based model-checking algorithms. The abstraction aims explicitly at the application in a multi-agent solving scenario where multiple instances of the same solver exchange information and, in this way, cooperate to solve a single problem instance. We instantiate the framework to obtain a parallel version of a successful pd-kind algorithm and experimentally show that exchanging information can significantly improve performance. Since pd-kind relies on interpolation as a sub-procedure, we use our novel interpolation algorithm to obtain more diverse behaviour of the agents, and this constitutes a large part of the performance improvement.

Acknowledgements

I would like to thank my advisors, Prof. Natasha Sharygina and Prof. Jan Kofroň, for introducing me to the beautiful world of formal verification and guiding and supporting me throughout my studies. During this time, I had opportunities to work with and speak to many wonderful and brilliant people, for which I am sincerely grateful. In particular, I would like to thank Dr. Antti Hyvärinen for our many insightful and fun discussions, Prof. Grigory Fedyukovich for many inspiring discussions and the opportunity to visit him in Tallahassee, and Dr. Leonardo Alt for mentoring me during my internship at Ethereum Foundation. My gratitude also goes to my group colleagues Sepideh Asadi, Masoud Asadzadeh, Dr. Matteo Marescotti, Rodrigo Otoni and Konstantin Britikov.

I gratefully acknowledge the financial support of the Swiss National Science Foundation (projects 200020_166288 and 200021_185031) and the Czech Science Foundation (projects 17-12465S and 20-07487S).

Lastly, I would like to thank my wife Ivana for supporting me and believing in me during my studies.

Contents

Chapter 1

Introduction

Nowadays, the presence of software in our lives is inescapable. The problems caused by software errors range from minor inconveniences to severe threats to human lives. Significant resources are therefore applied to eliminate such errors and ensure that programs conform to their specifications. *Testing* is the traditional method for detecting bugs in software systems. Different testing methods, from unit tests through integration tests to system tests, aim to detect problems in different phases of a development cycle. While testing is very good at quickly detecting common issues, it suffers from two deficiencies. Firstly, designing the tests requires a substantial *manual* effort. Secondly, testing can uncover errors but cannot prove their absence.[1] Moreover, testing often fails to cover the corner cases of the system under test. These deficiencies of testing led to the development of *semi-automated* and later *fully automated formal methods* that can also *prove* the *absence* of errors, not only their presence. Such techniques fall under the broad notion of *automated software verification.*

1.1 Automated Software Verification

Formal verification applies logic-based methods to prove, in a mathematical sense, that a system satisfies its specification. In the software domain, formal verification can be *interactive* or *automated.* In interactive verification, a human user guides the verification tool in its search for mathematical proof of correctness. Several popular large projects, such as Dafny [145], seL4 [134] or CompCert [147, 148] heavily rely on *interactive theorem provers* for verification. While effective, interactive verification can be very tedious and requires expert knowledge on the user's side to guide the tool successfully. On the other hand, automated formal verification aims to eliminate the human factor from the process altogether. The growing complexity of the systems being built nowadays requires automating the verification process as much as possible. For this reason, automated

[1]To quote Djikstra precisely: "Program testing can be used to show the presence of bugs, but never to show their absence!" [76]

software verification has attracted much attention in the last two decades, not only in the research community but also in the industry. Companies such as Microsoft, Amazon, Meta and Ethereum Foundation are investing in automated reasoning with the goal of providing formal proofs of correctness to their customers (see, e.g.,[53]). From the theoretical perspective, the undecidability of the halting problem [70] represents the fundamental barrier to what is possible to achieve in automated software verification. We cannot hope to find an algorithm that would work and terminate on every single problem instance of automated verification. Nevertheless, researchers from academia and industry are developing new techniques and heuristics to constantly push the frontier of what is possible in practice.

Logic plays a fundamental role in formal verification. The case for applying logic to formally *prove* correctness of programs has been made very early in the history of computers by such great names as Floyd, Hoare, Dijkstra and others [77, 97, 112]. *Hoare logic* [112] (also Floyd-Hoare logic) is a paradigm for program verification that is still widely used nowadays. The idea is to construct *Hoare triples*, annotations of the program statements of the form $\{P\}S\{R\}$ where S is the program statement, P is a precondition and Q is a postcondition. Given a program state that satisfies the precondition P, executing the statement S yields a state that satisfies the postcondition R. Hoare triples can serve as a proof of the correctness of a program. They are still used today, e.g., in a modern automata-based approach to software verification [110].

Another logic-based method for automated verification is *model checking* [64, 167]. In short, model checking consists of formally modelling a system and its specification—for example, with finite-state transition graphs and temporal logic formulas—and automatically deciding if the specification holds in the model using efficient *decision procedures*. Full automation, together with the ability to produce counterexamples (error traces) when the specification is not satisfied, has been the key ingredient in the success and broader adoption of model checking. Although theoretically model checking can verify finite-state systems completely, it was initially used mainly for bug-finding due to scaling issues. A great leap in scalability has been achieved by a switch from *explicit* graph representation to *symbolic* representation.[2] The advantage of symbolic reasoning is the ability to manipulate and reason about large *sets of states*, instead of one state at a time. At first, binary decision diagrams (BDDs) were used for the symbolic representation [46]. However, with the significant advancement in satisfiability solving in the last two decades, most symbolic model checkers now use logical formulas (propositional or first-order) to represent and reason about the problem.

The first approach relying purely on a SAT solver was *Bounded Model Checking* (BMC) [29]. BMC formulates the existence of a counterexample path of fixed length in the system as a satisfiability query. It iteratively increases the considered length until a counterexample is found. If an upper bound on possible lengths can be computed,

[2]Correspondingly, the original approach is now referred to as explicit-state model checking, while the latter is now known as symbolic model checking.

BMC can even prove the system safe by refuting the existence of a counterexample of any possible length. However, such an upper bound might not exist (in infinite-state systems) or might be prohibitively large in practice. Nevertheless, BMC has proven to be an excellent bug-finding technique and is still considered state-of-the-art even nowadays. A related technique, that aims to prove that a given (safety) property holds in all states of the system, is *k-induction* [186]. It also uses a BMC-style search for counterexamples, but it additionally attempts to construct an inductive proof of safety (using increasing induction depth). This technique is complete for finite-state systems if only loop-free paths are considered as potential counterexamples to induction. A different approach based on BMC that can prove the system safe is *Interpolation-based Model Checking* (IMC) [155]. IMC was the first algorithm to apply *Craig interpolation* [68] to compute *over-approximations* of reachable states and use the over-approximations in a fixed-point computation to prove that all reachable states satisfy the safety property. IMC popularized the concept of Craig interpolation for abstraction in the verification community, and this led to a large amount of research on interpolation procedures, as well as to several new verification algorithms that rely on Craig interpolation to a lesser or greater extent [2, 4, 129, 157, 176, 177, 183]. However, the concept of abstraction in verification is more general, and its importance was recognized even before Craig interpolation was introduced to the verification community. Abstraction, in general, means deliberately ignoring properties of the system deemed unimportant for the property that should be proven. It reduces the number of states in the model, preventing the state-space explosion. If the abstract system is proven safe, the original system is also safe. However, abstraction may introduce spurious behaviour. If an abstract counterexample is found but does not correspond to a feasible behaviour of the original system, the abstraction must be *refined* to exclude the spurious counterexample. *Predicate abstraction* [100] is a common technique that abstracts a program using a fixed set of predicates over the program variables. The construction of an abstract system is fully automatic; however, initially, it could not refine the abstraction automatically. This missing piece was provided by *Counterexample-guided abstraction refinement* (CEGAR) framework [63]. CEGAR automatically analyzes spurious counterexamples and refines the abstraction to rule out the infeasible path. Further improvements were achieved with *Lazy Abstraction* [111], which refines the abstraction on demand instead of starting the analysis of the refined system from scratch. Craig interpolants have also been successfully applied in the context of lazy abstraction to discover relevant predicates automatically [157].

As mentioned before, many of the new model-checking techniques were enabled by the incredible advancement in SAT solving [30]. However, in an orthogonal direction, the advancement in SAT solving also fueled advancement in *Satisfiability Modulo Theories* (SMT) [17]. New efficient SMT solvers allowed the researchers to lift many model-checking techniques, initially developed for hardware and relying on SAT solvers, to software verification. The application of k-induction for software verification has been studied, e.g., in [21, 43, 79, 80]. The combination of predicate abstraction and CEGAR is the core solving algorithm in the ELDARICA Horn solver [117]. Interpolation-based algorithms for the analysis of software were proposed, e.g., in [157, 158, 159]. A thorough comparison of several SMT-based software verification techniques—BMC, k-induction, predicate abstraction and lazy abstraction with interpolants—on a large set of C programs, together with an overview of related work, can be found in [22].

Another breakthrough in symbolic model checking occurred with the introduction of an algorithm called IC3 [41], later generalized to an approach dubbed *Property Directed Reachability* (PDR) [85]. One of the key features of the algorithm is that, unlike the previous algorithms, it does not require unrolling the transition relation. Relatively quickly after its introduction, it has become the dominant approach in hardware verification. The algorithm has been studied thoroughly [104], and various modifications have been proposed [18, 82]. Similar to other hardware model-checking algorithms, IC3/PDR has also been lifted and applied in the domain of software [58, 59, 113]. Soon after its introduction, PDR was generalized from transition systems to non-linear fixed-point operators, opening the door for efficient compositional analysis of programs with (recursive) functions [113]. Continuing in this direction, SPACER algorithm [137] introduced two novel aspects: under-approximating summaries and *Model-based Projection* (MBP). Under-approximating summaries serve as caches of truly reachable states, which allow the algorithm to avoid repeated work. MBP enables efficient computations of predecessors by under-approximating quantifier elimination. The combination of CEGAR and IC3/PDR, named *Counterexample to Induction-guided Abstraction Refinement* (CTIGAR), has been proposed in [31]. *Property-directed k-induction* (PD-KIND) successfully replaced the inductive reasoning in IC3/PDR with k-inductive reasoning [129].

1.1.1 Modelling Software Verification Problems as Constrained Horn Clauses

The earlier discussion highlighted that there is a large body of *logic-based* algorithms for automated verification. These algorithms eventually represent the problem using logical formulas and reduce subtasks to satisfiability checks, decided by underlying SAT or SMT solver. However, software verifiers that implement these algorithms are typically developed for a single domain, and non-trivial effort is required to successfully apply a concrete algorithm in a concrete domain. Much of this work is repeated when the same algorithm is applied in a new domain (such as a new programming language).

To overcome this problem hindering developments of software verification tools, the framework of *Constrained Horn Clauses* (CHC) has been proposed as a unified, purely logic-based, intermediate format for reasoning about software verification tasks [101]. Originally named *Horn-like clauses* [101], they use the language of logical constraints to capture various verification tasks (e.g., safety, termination and loop invariants computation) from different domains such as transition systems, functional programs, procedural and recursive programs, concurrent and distributed systems [101, 105, 118]. The advantage of CHC representation is that it nicely separates the task of modelling a verification problem from the actual solving. It represents an application of a well-known and

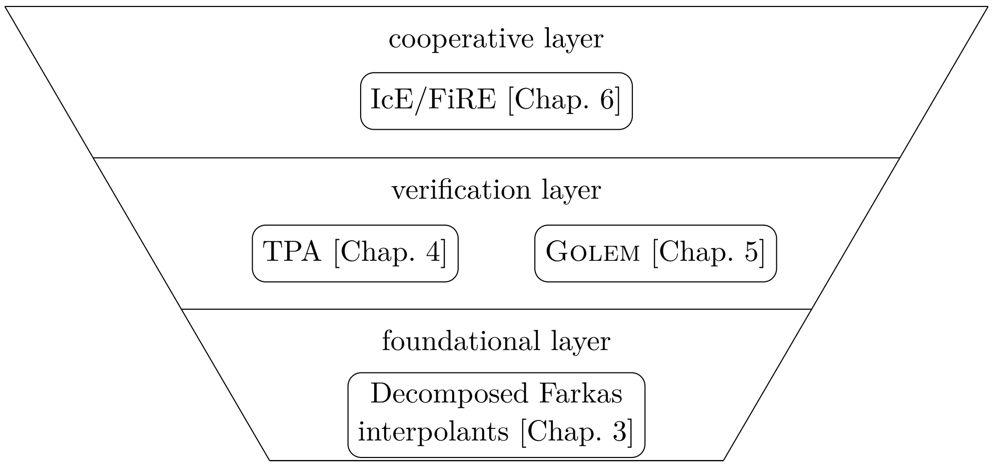


Figure 1.1. Layered approach to CHC solving and our contributions

important principle in software design—*separation of concerns.* It avoids repeated engineering effort: a specialized CHC solver can be re-used for different verification tasks across domains and programming languages. On the side of the front end, the main task becomes the translation of the source code to the language of constraints. Several CHC-based verification frameworks have already been developed, for example, SeaHorn for C/C++ [107], JayHorn for Java [132], RustHorn for Rust [154], HornDroid for Android [48], SolCMC for Solidity [5, 153]. At the back end, developing the solver is freed from the complexities and peculiarities of a given application domain. It can focus on a well-defined formal problem—satisfiability of a system of constrained Horn clauses. Many techniques developed in the context of model checking and software verification have been lifted to the uniform setting of CHC. They are now implemented in the specialized Horn solvers, for example, Spacer [137], Eldarica [117, 175], FreqHorn [93, 94], HoIce [52] and others. Horn solvers now compete in an annual international competition CHC-COMP[3] on a large set of benchmarks from various domains.

1.2 Challenges and Contributions

As mentioned in the previous section, many interesting problems in automated software verification can be reduced to the problem of the satisfiability of a system of constrained Horn clauses. However, this means that one has to pay the price for the powerful formalism of CHC: satisfiability of CHC over most interesting theories (e.g., linear real or integer arithmetic) is undecidable [119, 126, 137, 176]. Thus, we cannot hope to find a single algorithm to solve all instances of CHC satisfiability: there is no "holy grail" that we could hope to discover. Nevertheless, there are compelling reasons for developing techniques and tools for CHC solving. On the theoretical side, there are specific subclasses

[3]`https://chc-comp.github.io`

of CHC satisfiability that are decidable, for example, CHC over propositional logic or recursion-free CHC over linear integer arithmetic [137, 176]. On the practical side, many interesting instances from industry *can* be solved very efficiently. Researchers are continuously coming up with new improvements or even whole new algorithms to attack the classes of problems not solvable before with existing resources. However, many problems, especially related to scalability, persist.

In this thesis, we show how the challenge of automated software verification can be approached on multiple levels and we present contributions on all layers. The layered approach and our contributions are depicted on Figure 1.1. The *foundational layer* represents decision and interpolation procedures which constitute the backbone of logic-based verification algorithms. The *verification layer* represents sequential verification algorithms which verify software systems against their specification. On the *cooperative layer*, multiple agents cooperate on solving a single instance of the verification problem. While research has been conducted on all layers, it is often narrow in scope, focusing only on one layer at a time. For example, Satisfiability Modulo Theories (SMT) solvers, which implement decision and interpolation procedures for various theories of first-order logic, are often used as black boxes in traditional software verifiers. While possible, we argue that such strict separation does not unlock the full potential of SMT-based software verification. In our research we promote a *tight integration* of the verification algorithm and the underlying SMT solver. This integration is quite natural in the CHC framework; it can be found in the best CHC solvers ELDARICA and SPACER. Similarly, on the cooperative layer, the large amount of computational power available nowadays with multi-core and cloud computing can be easily utilized to run multiple verification algorithm in parallel with a *portfolio* approach. However, only full understanding of the sequential algorithms enables *information exchange* between the individual agents and consequently *cooperative learning* for all agents. Multi-agent solving with cooperative learning achieves greater improvement over sequential algorithms than a pure portfolio. In our work, we have identified challenges across all three layers and proposed solutions that are the contributions of this thesis.

At the foundational layer, we studied interpolation procedures for conflicts in linear real arithmetic (LRA). This is a core subprocedure in the computation of interpolants in general problems, not only in linear real arithmetic but also in linear integer arithmetic. As such, it can significantly affect the performance of any interpolation-based model-checking algorithm that analyzes systems with discrete or continuous behaviour. This work was motivated by the problems at the verification layer, where interpolation-based verification algorithms relying on existing interpolation procedure were sometimes diverging even on relatively simple programs. We have developed and implemented a new interpolation algorithm for LRA to address this problem.

At the verification layer, we have developed a novel concept of *transition power abstraction* and created a model-checking algorithm that builds on this idea, utilizing interpolation and SMT solving. This technique addresses the apparent scalability issue in existing model-checking techniques when dealing with systems with deep counterexamples, i.e., systems that exhibit faulty behaviour only after a very long time. Transition power abstraction automatically computes abstract transitions that guide the search for faulty behaviour. Additionally, it can be used to prove the absence of such faulty behaviour by discovering safe transition invariant of the system. Another contribution at the verification layer is our new efficient Horn solver Golem. Golem is tightly integrated with the interpolating SMT solver OpenSMT [122] and offers several model-checking algorithms as the back-end reasoning engine, including our novel algorithms based on the transition power abstraction. It is implemented in C++ and publicly available on GitHub[4].

At the cooperative layer, we have formalized an abstract framework for induction-based reasoning suitable for cooperative multi-agent solving with information exchange. Leveraging the SMTS infrastructure for distributed constraint solving [152], we have implemented a parallel version of the pd-kind algorithm [129] as an instance of our proposed multi-agent architecture. Next, we describe each of the contributions in more detail.

1.2.1 Decomposed Farkas Interpolants

Craig interpolants are key ingredients in interpolation-based model-checking algorithms, used for computing abstractions or as candidates for inductive invariants. In the theory of linear arithmetic—required for modelling any numerical properties of a system—an interpolant is typically computed based on Farkas' lemma [90]. Such an interpolant, called *Farkas* interpolant, is always a *single* inequality. In model-checking, this property is not always desirable and can even lead to divergence of the model-checking algorithm [179, 190]. This problem can be fixed on the level of the model checker, for example, by globally monitoring progress and applying specialized techniques when a diverging behaviour has been detected [190]. Alternatively, the model checker can apply *Interpolation abstraction* [177] to restrict interpolants to conform to a prescribed form. A different approach is to keep the model-checking algorithm unchanged and attempt to fix the problem at the foundational layer by modifying the interpolation procedure. A specialized interpolation procedure ensuring *finite convergence property* has been proposed to address this divergence problem in model checking [179]. The disadvantage of these solutions is that they are rather intrusive. They either introduce complexity to the model-checking algorithm [177, 190] or require a less efficient decision procedure than the one used in state-of-the-art SMT solvers [179].

A mechanism for controlling the strength of an arithmetic interpolant that is not intrusive has been proposed in [8]. However, it can only compute interpolants *weaker* than Farkas interpolant (and stronger than its dual). In Chapter 3, we propose an interpolation procedure for theory conflict over linear arithmetic that uses the methods from linear

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

algebra to identify independent components in the Farkas coefficients and *decompose* the Farkas interpolant into multiple linear inequalities. Our approach is simple yet powerful. Similar to [8], it is not intrusive. It only requires minor changes in the interpolation procedure, where the proof of unsatisfiability in the form of Farkas coefficients is analyzed to discover linearly independent components. Decomposition to multiple independent components yields a *decomposed* Farkas interpolant in the form of a *conjunction* of linear inequalities. The decomposed interpolant always implies the Farkas interpolant, i.e., it is *logically stronger*. Stronger interpolants can be beneficial, for example, to compute tighter abstractions. We show that decomposed interpolants can prevent divergence of the model-checking algorithm. The results presented in Chapter 3 were published in the proceedings of TACAS 2019 [37] and in the extended journal publication [38].

1.2.2 Transition Power Abstraction

In this line of research, we investigated transition systems, a common formal model for hardware and software. Transition systems are characterized by a set of initial states, a set of bad states and a transition relation that defines reachability in the system. The safety problem in transition systems translates to the problem if some error state is reachable from the initial states in a finite number of steps of the transition relation.

Existing model-checking algorithms struggle to scale when faced with unsafe systems where only *deep* counterexamples exist. Deep counterexamples represent behaviours of the system that reach an error state after a large number of transition steps, roughly in the order of thousands. The reason existing algorithms struggle to scale is the inherent slow increment of the *safety horizon*—a number of steps for which an algorithm in the current run has already proved that no counterexample of that length exists.

The concept of *transition power abstraction* (TPA) is a solution to this problem. The idea is to maintain a sequence of transition formulas where each element over-approximates twice as many steps as its predecessor. This doubling abstraction enables doubling the safety horizon with a single SMT query if the current abstraction is sufficiently precise. With this ability, the algorithm can quickly rule out the existence of short counterexamples and focus the search on the deeper parts of the state space. The key point is that all elements of the TPA sequence are quantifier-free transition formulas, i.e., containing only two copies of the state variables. This keeps the satisfiability queries representing the existence of (abstract) paths manageable for the underlying SMT solver. Craig interpolation plays a crucial role in the computation and refinement of the TPA sequence.

With the TPA sequence, it is possible to quickly reach far greater depths in the system than traditional algorithms. However, it can also serve as a source of candidates for *transition invariants*. Transition invariants over-approximate unbounded reachability in the system, regardless of the initial or bad states. Intuitively, if a state s_1 can reach state s_2 in some number of transition steps, then the pair (s_1, s_2) satisfies the relational property defined by the transition invariant. While transition invariants have been studied in connection to termination and other liveness properties [141, 164, 165], they can also be used to prove safety.

Chapter 4 presents a novel model-checking algorithm based on the TPA sequence. We show how the construction and refinement of the TPA sequence are elegantly interwoven with checking the existence of bounded paths from initial to bad states. The experiments on a set of challenging safety problems of multi-phase loops show a substantial improvement in detecting deep counterexamples. In proving safety, the ability of the proposed algorithm is orthogonal to other model-checking techniques. These results were published in the proceedings of TACAS 2022 [36] and in the proceedings of FMCAD 2022 [35].

1.2.3 The Golem Horn Solver

In Chapter 5, we present Golem, a new efficient Horn solver for CHCs over linear real and integer arithmetic. It can serve as a research tool for prototyping new ideas related to CHC solving and as a powerful back end for domain-specific verification tools. The key features of Golem are:

- **Tight integration with SMT solver** Golem is tightly coupled with the underlying interpolating SMT solver OpenSMT. Tight coupling brings several advantages for Golem. The main advantage is full control over the solving and interpolation processes in OpenSMT. From the engineering point of view, Golem re-uses mature and efficient data structures of OpenSMT for term representation and manipulation. This saves development time and makes Golem more efficient and less error-prone.

- **Modular architecture** The process of deciding CHC satisfiability is separated into normalization, preprocessing and actual solving. The normalization creates an internal representation of a CHC system in the form of a labeled multi-hypergraph which is the representation on which the transformations in preprocessing and the solving algorithms (back-end engines) operate. Preprocessing consists of several transformations that take as input the graph representation and output a modified graph. These transformations are similar to the idea of optimization passes on intermediate representation in compilers. The final graph of the preprocessed system is passed to the model-checking algorithm in the chosen back-end engine.

- **Multiple back-end solving engines** The aim from the beginning of the development was to implement multiple, mostly interpolation-based, algorithms for solving CHC systems. Golem currently supports five different back-end engines: Bounded Model Checking [29], k-induction [186], Lazy Abstraction with Interpolants [157] (also known as Impact), Spacer [137], and TPA (Chapter 4). The modular architecture and tight integration with the SMT solver enable easy prototyping of new algorithms as the infrastructure of Golem already provides procedures to handle many subtasks, and the engine developer can focus on the algorithm itself.

Golem has been crucial for developing and evaluating the TPA algorithms proposed in this thesis. It was used in the experiments reported in TPA publications [35, 36]. It has also successfully participated in the international CHC competition (CHC-COMP) in 2021 and 2022: It took second place in LRA-TS track and third place in LIA-Lin track in 2021. In 2022, it beat all solvers except the non-competing Spacer in the tracks LRA-TS, LIA-Lin and LIA-nonlin. An official tool paper for Golem is currently under preparation.

1.2.4 The IcE/FiRE Framework for Cooperative Model Checking

Given that the problem we are trying to solve is undecidable in general, it is natural that different solving techniques exhibit different strengths and weaknesses on various instances of the problem. The straightforward way to harness the strengths of various approaches is to run them in parallel in a *portfolio* manner [121]. The portfolio approach represents concurrent and independent execution of multiple *agents* (different tools, different configurations of the same tool) on the same problem instance. The advantage of the portfolio is that no changes are required in the participating agents. However, there is no communication *between* the agents, hence no cooperation, no learning from each other.

A potentially better approach is a *cooperative* one, where the agents *share* the knowledge they acquire during the solving process. The knowledge sharing in the form of *lemmas* has been successfully applied in SMT solving [151, 199], as well as in IC3-based model checking [50, 150]. A push for cooperative solving is also present in the software verification community [24, 27].

In Chapter 6, we generalize the concepts of recently developed induction-based model-checking algorithms, especially pd-kind [129]. We propose an abstract IcE/FiRE framework, whose instances can easily participate in cooperative parallel solving. Our contribution is the theoretical IcE/FiRE framework and its concrete instantiation yielding a parallel pd-kind algorithm. Using SMTS [152], a framework for distributed solving, we experimentally show the viability and usefulness of parallel pd-kind. The results show the importance of sharing information between solvers. Moreover, as pd-kind is an interpolation-based algorithm, the experiments also show that using both Farkas interpolants and decomposed Farkas interpolants from Chapter 3 is an important source of diverse behaviour that leads to much better performance of our parallel solver. The results presented in Chapter 6 were published in the proceedings of VMCAI 2020 [39].

1.3 Organization of This Thesis

After this Introduction, Chapter 2 gives an overview of the main notions we use throughout this thesis, including Satisfiability Modulo Theories (SMT), interpolation, constrained Horn clauses, and transition systems. The following chapters present our contributions to automated software verification. Chapter 3 presents our main contribution at the foundational layer: new interpolation algorithm for conflicts in the theory of linear arithmetic.

Chapter 4 presents our new model-checking algorithm based on the concept of Transition Power Abstraction (TPA) sequence. Chapter 5 describes our next contribution at the verification layer: the Horn solver Golem. Our final contribution, the IcE/FiRE framework for cooperative parallel verification, is given in Chapter 6. We conclude in Chapter 7 with final remarks and outline future work based on the results presented in this thesis.

1.4 Publications Overview

1.4.1 Publication in Thesis

Much of the material presented in this thesis was published at conferences and in journals as listed below:

- **Blicha, M.**, Hyvärinen, A. E. J., Kofroň, 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*, Springer International Publishing, Cham, pp. 3–20.

- **Blicha, M.**, Hyvärinen, A. E. J., Marescotti, M. and Sharygina, N. [2020]. A cooperative parallelization approach for property-directed k-induction, in D. Beyer and D. Zufferey (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer International Publishing, Cham, pp. 270–292.

- **Blicha, M.**, Hyvärinen, A. E. J., Kofroň, J. and Sharygina, N. [2022]. Using linear algebra in decomposition of Farkas interpolants, *International Journal on Software Tools for Technology Transfer* **24**(1): 111–125.

- **Blicha, M.**, Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2022]. Transition power abstractions for deep counterexample detection, in D. Fisman and G. Rosu (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 524–542.

- **Blicha, M.**, Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2022]. Split transition power abstractions for unbounded safety, in A. Griggio and N. Rungta (eds), *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design - FMCAD 2022*, TU Wien Academic Press, pp. 349–358.

1.4.2 Additional Publications

In addition to our main results, we participated in related projects whose results were also published at conferences:

- Asadi, S., **Blicha, M.**, Fedyukovich G., Hyvärinen, A. E. J., Even-Mendoza K., Sharygina N. and Chockler H. [2018]. Function Summarization Modulo Theories, in G. Barthe, G. Sutcliffe and M. Veanes (eds), LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, vol 57, pp. 56–75.

- Marescotti, M., **Blicha, M.**, Hyvärinen, A. E. J., Asadi, S. and Sharygina, N. [2018]. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In: T. Margaria, B. Steffen (eds), *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice.* ISoLA 2018. Lecture Notes in Computer Science, vol 11247. Springer, Cham.

- Asadi, S., **Blicha, M.**, Hyvärinen, A. E. J., Fedyukovich, G. and Sharygina, N. [2020]. Farkas-Based Tree Interpolation. In: D. Pichardie, M. Sighireanu (eds), *Static Analysis.* SAS 2020. Lecture Notes in Computer Science, vol 12389. Springer, Cham.

- Asadi, S., **Blicha, M.**, Hyvärinen, A. E. J., Fedyukovich, G. and Sharygina, N. [2020]. Incremental Verification by SMT-based Summary Repair. In A. Ivrii and O. Strichman (eds), *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design - FMCAD 2020*, TU Wien Academic Press, pp. 77–82.

- 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)*, San Francisco, CA, USA, pp. 541–546.

- **Blicha, M.**, Kofroň, J. and Tatarko, W. [2022]. Summarization of Branching Loops. *In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22).* Association for Computing Machinery, New York, NY, USA, pp. 1808–1816.

- Alt, L., **Blicha, M.**, Hyvärinen, A. E. J. and Sharygina, N. [2022]. SolCMC: Solidity Compiler's Model Checker. In: S. Shoham, Y. Vizel (eds), *Computer Aided Verification. CAV 2022.* Lecture Notes in Computer Science, vol 13371. Springer, Cham.

1.5 Cotutelle de Thèse

Cotutelle de thèse enables PhD students to conduct research and study at two universities in different countries—in this case, Università della Svizzera italiana in Lugano and Charles University in Prague. For cotutelle de thèse, the doctoral candidate has to be enrolled in and fulfil the requirements of both universities. The supervision of this PhD thesis is shared between Prof. Natasha Sharygina at Università della Svizzera italiana, and Prof. Jan Kofroň at Charles University. The defense of the thesis takes place at Università della Svizzera italiana according to the terms of the agreement. The dissertation committee has been chosen to satisfy the respective requirements of both universities.

Chapter 2

Preliminaries

2.1 Satisfiability Modulo Theories

The question of determining the satisfiability of a boolean formula—the *SAT* problem—is one of the most famous problems in computer science. The formulation of the problem is simple: Given a boolean formula, decide if there exists an assignment of its variables under which the formula evaluates to true ($\top$). However, *solving* the SAT problem is complex. It was the first problem shown to be an NP-complete problem, a result now known as Cook-Levin theorem [66, 188]. Despite its theoretical complexity, there have been extensive studies of algorithms for solving the SAT problem. The reason has been mostly pragmatic; many interesting problems can be efficiently encoded to SAT, including problems from hardware and software verification. However, problems from the verification domain often require (or at least benefit from) a stronger modelling language, such as first-order logic. Typically, the full power of first-order logic is not required; only satisfiability with respect to some *background theory* needs to be decided. This observation led to the development of specialized *decision procedures* that decide the satisfiability of formulas in a fragment of first-order logic corresponding to a specific theory. The research field concerned with this problem is called *Satisfiability Modulo Theories* (*SMT*) [14]. Following the SAT terminology, the procedures for solving the SMT problem are called *SMT solvers*. After initial independent efforts, an international initiative for standardization and benchmark collection called SMT-LIB was formed in 2003 to facilitate research and development in SMT [15]. The SMT-LIB initiative later established SMT workshop, an international workshop for connecting SMT developers and users, and SMT-COMP, an international competition of SMT solvers supporting the SMT-LIB input format.

We refer the reader to the excellent textbooks, e.g. [17, 142], for a detailed overview of the field, architecture of the solvers and specific decision procedures. The chapter on *Satisfiability Modulo Theories* [17] in the *Handbook of Satisfiability* [30] gives an excellent general overview; the textbook *Decision Procedures* [142] focuses on decision procedures for various theories. Here, we only give the necessary terminology used throughout the thesis.

Syntax

We work in fragments of first-order logic defined by a *signature* Σ, a set of *predicate* and *function* symbols with an associated *arity*. Besides the signature, the logic uses the standard logical *connectives* $\wedge$,$\vee$,$\neg$,$\rightarrow$,$\leftrightarrow$ (conjunction, disjunction, negation, implication, equivalence, respectively). A *term* is a variable or a function symbol applied to terms (respecting the symbol's arity). An *atomic* formula (or simply *atom*) is a predicate symbol applied to terms (respecting the symbol's arity). It is very convenient in SMT to view the logic as *multi-sorted* logic, where variables are associated with a specific *sort* or *type*. The prominent sort is `Bool` associated with the *Boolean* expressions. Similarly, the function and predicate symbols prescribe sorts for their arguments; additionally, function symbols have an *return sort*, while this is implicitly the sort `Bool` for predicate symbols. Variables are sometimes viewed as 0-ary constant symbols with `Bool` variables as 0-ary predicate symbols and other variables as 0-ary function symbols. `Bool` variables also count as atomic formulas. SMT assumes that the equality symbol $=$ is part of the signature for every theory and is always interpreted as proper equality. A *literal* is either an atom or its negation. A *clause* is a disjunction of literals. A formula is in *conjunctive normal form* (CNF) if it is a conjunction of clauses. SAT solvers take as input a propositional formula in CNF, and SMT solvers often internally translate the input formula to CNF. In this thesis, we work, for the most part, with *quantifier-free* formulas.

Semantics

The goal in SMT is to decide *satisfiability* of a given formula. A *model* $\mathcal{M}$ for a signature Σ consists of a non-empty set A called the *universe* of the model (or a set for each sort in case of multi-sorted logic) and an interpretation for the symbols from Σ, which maps function symbols to functions over the universe and predicate symbols to relations over the universe. The value of terms and formulas in the given model is defined inductively, as usual (see, e.g., [17]). A formula φ is satisfiable if there exist a *model* $\mathcal{M}$, where it evaluates to *true*, i.e., $\mathcal{M} \models \varphi$. In SMT, the satisfiability is decided with respect to some background theory $\mathcal{T}$, which specifies the interpretation of the symbols of Σ. Then the model only has to specify the interpretations of the formula's variables. A formula is $\mathcal{T}$-valid if it evaluates to true in all models of $\mathcal{T}$. A clause c that is $\mathcal{T}$-valid is also called a $\mathcal{T}$-lemma. When the theory is known from the context or the context is independent of a particular theory, we often refer to $\mathcal{T}$-lemmas simply as *theory* lemmas. In this thesis, we work with the theory of linear arithmetic.

2.1.1 Linear Real and Integer Arithmetic

One of the essential theories (not only in software verification) is *linear arithmetic*. The following is the grammar from [142] for the conjunctive fragment of linear arithmetic.

Definition 2.1 (linear arithmetic [142])**.** *The syntax of a formula in* linear arithmetic *is defined by the following rules:*

$$\begin{aligned}
\mathit{formula}&\colon \mathit{formula} \wedge \mathit{formula} \mid (\mathit{formula}) \mid \mathit{atom}\\
\mathit{atom}&\colon \mathit{sum}\ \mathit{op}\ \mathit{sum}\\
\mathit{op}&\colon = \mid \leq \mid <\\
\mathit{sum}&\colon \mathit{term} \mid \mathit{sum} + \mathit{term}\\
\mathit{term}&\colon \mathit{identifier} \mid \mathit{constant} \mid \mathit{constant}\ \mathit{identifier}
\end{aligned}$$

We consider linear arithmetic over the domain of integers and the domain of reals. In the former case, we talk about *linear integer arithmetic* (LIA), and in the latter, about *linear real arithmetic* (LRA). In the syntax of Definition 2.1, *identifiers* correspond to *variables*, while *constants* are mathematical integer constants. LIA corresponds, up to a syntactic sugar, to the theory of *Presburger arithmetic* with the signature $(0, 1, +, -, \leq)$ and the usual interpretation of the symbols [17]. For example, multiplication by a positive constant is just a sum, $nx = \overbrace{x + \ldots + x}^{n \text{ times}}$. Minus operation can also be expressed within the syntax above: $x - y$ can be written as $x + -1y$.

Quantifier-free fragments of both LRA and LIA are decidable. While the conjunctive fragment of LRA is decidable in polynomial time, the conjunctive fragment of LIA is NP-complete. Interestingly, even though polynomial algorithms for deciding LRA exist, most SMT solvers use decision procedures based on the *Simplex* algorithm [84]. Simplex is exponential in the worst case but typically very fast on real-world problems where the exponential behaviour is rarely observed. Decision procedures for LIA in SMT solvers typically follow the *branch-and-bound* or *branch-and-cut* paradigm. First, the relaxed version of the problem is solved (as if with the domain of reals). If a solution for the relaxed problem yielded a non-integer solution for some variable, new constraints are added to exclude the non-integer solution but preserve the set of integer solutions. For more details on decision procedures for linear arithmetic and their implementation in SMT solvers, we refer the reader to [34, 44, 78, 84, 102, 142].

2.2 Craig Interpolation

Craig interpolant for a valid implication $A \rightarrow C$ is a formula I such that $A \rightarrow I$, $I \rightarrow C$ and all free variables of I occur both in A and C. In verification, an alternative formulation, obtained by replacing C with $\neg B$, is more common. Then the validity of the implication $A \rightarrow C$ is equivalent to the unsatisfiability of $A \wedge B$. The rationale is that A represents the feasible behaviour of a system and B represents the error behaviour. In this case, the interpolant over-approximates *safe* behaviour of the system.

Craig [68] showed that in the first-order logic an interpolant always exists for any unsatisfiable pair A and B. However, for practical applications, it is crucial to compute the interpolants efficiently, and quantifier-free if possible. Interpolating SMT solvers OpenSMT [122], MathSAT [60] and SMTInterpol [56] compute interpolants from a proof of unsatisfiability. In propositional logic, an algorithm that computes interpolant from a resolution proof in linear time with respect to the size of the proof was introduced by Pudlák [166], Krajíček [139] and Huang [120] (note, however, that the proof itself might be exponential with respect to the size of the formula). The algorithm traverses the resolution proof, computing partial interpolant for each clause derived in the proof. The partial interpolant for the clause $\perp$ is an interpolant for $A \wedge B$. Another algorithm based on the same proof traversal has been introduced by McMillan [155] and both have been later showed to be an instantiation of the general framework of *labeled interpolation systems* [81]. Further study of interpolation procedures in propositional context include, e.g., proof manipulation for interpolant generation [170, 172], proof-sensitive interpolation procedure [4, 6], generalization of the labeled interpolation systems [197], interpolant computation from DRUP and DRAT proof systems [109, 169].

Moving from propositional logic to the first-order logic, modern SMT solvers based on the framework known as CDCL($\mathcal{T}$) or $\mathcal{T}$-DPLL [17] combine propositional interpolation algorithms with theory-specific interpolation procedures for theory lemmas. During the search, a CDCL($\mathcal{T}$)-based SMT solver blocks satisfying assignments of the boolean skeleton of the formula by generating $\mathcal{T}$-lemmas falsified by the current assignment. These $\mathcal{T}$-lemmas generated on-the-fly are included in the resolution proof as a new type of leaves, besides the input clauses. A theory-specific interpolation procedures, *theory interpolators*, compute interpolants for each theory lemma; these are then treated as partial interpolants for the corresponding proof leaves in the standard propositional interpolation procedures. The combination of the propositional and theory-specific part of the interpolation algorithm has been described in [200]. More recent work on proof-preserving interpolation that does not restrict the SMT solver has been given in [55, 57]. Theory-specific interpolation procedures have been given for the theory of equality and uninterpreted functions [7, 99, 156], linear arithmetic over rationals [8, 61, 156] and integers [102, 103], and arrays [45, 114]. Interestingly, some theories, such as linear integer arithmetic or theory of arrays with extensionality, do not admit quantifier-free interpolation, i.e., there are instances of an interpolation problem with quantifier-free A and B where all possible interpolants contain quantifiers. However, these theories can be extended in a simple way to get a theory that admits quantifier-free interpolation. This is the case for the theory of arrays with *diff* predicate [45] and for the theory of linear integer arithmetic extended with a ceiling function [103] or division-by-constant functions [55].

For many applications in model checking, the simple *binary* interpolant is not sufficient; instead, a property of a *collection* of interpolants is required [108]. Examples of such properties are path interpolation [128, 194], tree interpolation [55, 160, 182] and simultaneous abstraction [127]. Some interpolation procedures guarantee these properties for a collection of interpolants computed from a *single* proof of unsatisfiability, which is important for efficiency in interpolation-based model-checking algorithms [55, 108].

Path interpolation is of a particular interest in the context of this thesis. Our Horn solver Golem, which we describe in Chapter 5, uses path interpolation in its engine implementing Lazy Abstraction with Interpolants [157] and also for recomputation of satisfiability witnesses required due to preprocessing transformations. The following definition of path interpolation is taken from [128]. It also appeared under the name *interpolation sequence* [194]. Given an inconsistent formula $\varphi_1 \wedge \ldots \wedge \varphi_n$, a sequence of formula $I_0, \ldots, I_n$ is a *path interpolant* iff

- $I_0 = \mathit{true}$ and $I_n = \mathit{false}$,
- for all $1 \leq i \leq n$, $I_{i-1} \wedge \varphi_i$ implies I_i,
- for all $1 \leq i < n$, I_i uses only the common symbols of φ_i and φ_{i+1}.

2.3 OpenSMT Solver

OpenSMT is the in-house SMT solver of our group, Formal Verification and Security Lab, at USI, Lugano, Switzerland. It is open-source software available at GitHub.[1] OpenSMT is currently one of the best SMT solvers for quantifier-free linear real and integer arithmetic (QF_LRA and QF_LIA) according to the results from SMT-COMP [196], an annual competition between SMT solvers.[2] It was the best-performing competing solver for the logic QF_LRA in the single-query track in 2020–2022 and for the logic QF_LIA in 2022.

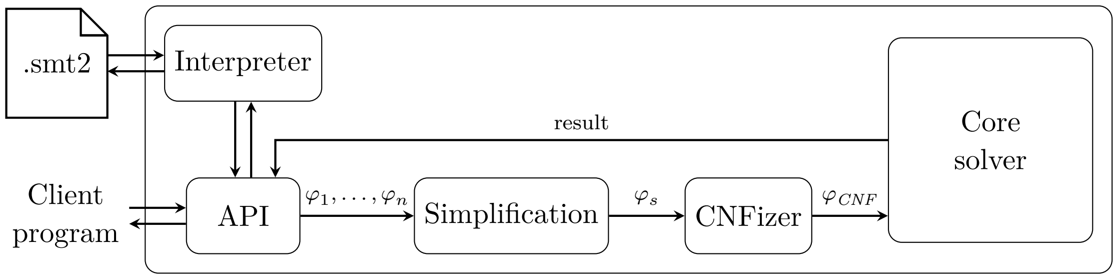


Figure 2.1. High-level architecture of OpenSMT

The high-level architecture of OpenSMT is depicted in Figure 2.1. It is the same as described in the last publication on OpenSMT [122]. Interaction with OpenSMT is available either through SMT-LIB scripts [15] or its application programming interface (API). The problem is given to the solver as a sequence of SMT formulas $\varphi_1, \ldots, \varphi_n$ (*asserted* to the solver, in the terminology of SMT-LIB), and the goal is to decide if the conjunction of these formulas is satisfiable. OpenSMT first applies general and theory-specific preprocessing that yields an equisatisfiable formula φ_s. Then, the formula

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

[2] `https://smt-comp.github.io/`

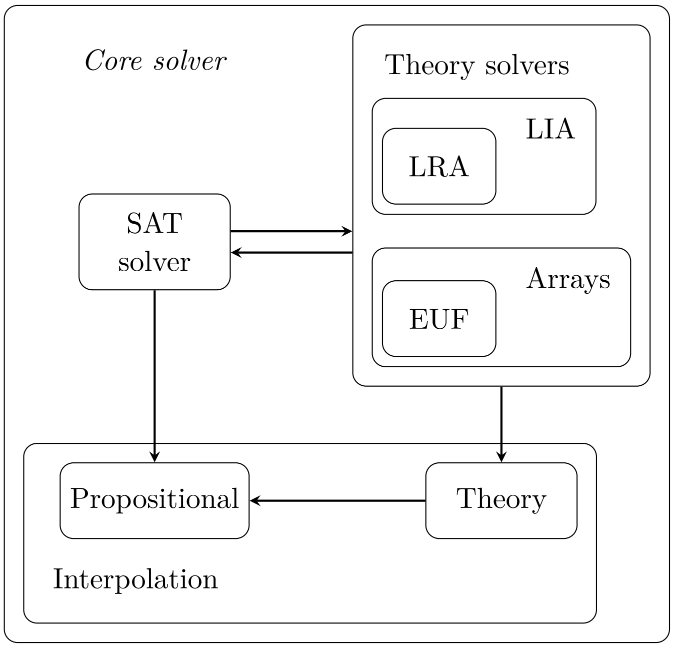


Figure 2.2. Detailed view of OpenSMT's core solver

is translated into a conjunctive normal form (CNF) and passed to the *core* solver. The core solver in OpenSMT follows the CDCL($\mathcal{T}$) framework (also known as $\mathcal{T}$-DPLL) [17]. Figure 2.2 depicts its internal architecture. The core consists of an augmented CDCL-based SAT solver connected to theory solvers able to decide the satisfiability of a conjunction of theory literals. The SAT solver searches for a satisfiable assignment of the propositional abstraction of the formula, and the theory solvers check the consistency of the current assignment in the theory $\mathcal{T}$. If the current assignment represents a $\mathcal{T}$-conflict, the theory solvers produce a $\mathcal{T}$-lemma which is falsified by the current assignment. When the theory lemma is added to the SAT solver, it backtracks and tries to find a different assignment. When the SAT solver finds a full propositional assignment consistent with the theory, the core solver finishes and reports the formula as satisfiable. If the SAT solver derives the empty clause, the core solver reports the formulas as unsatisfiable. Besides theory lemmas, the theory solvers typically apply theory propagation to derive literals that must be true under the current (partial) assignment. Additionally, they may produce new clauses and atoms that the SAT solver must decide before theory consistency can be checked. The SAT solver in OpenSMT's core solver is based on MiniSAT 2.0 [86], and it has theory solvers for deciding the theories of uninterpreted functions [74], linear real and integer arithmetic [78, 84] and arrays [54].

The interpolation module of OpenSMT follows the CDCL($\mathcal{T}$) split to the propositional and the theory part. When requested, the SAT solver in the core keeps track of the resolution chains that derive new clauses. From this information, the interpolation module reconstructs a detailed resolution proof and uses standard propositional interpolation procedures to derive an interpolant from the proof. OpenSMT implements the framework of Labeled Interpolation Systems (LIS) [81] for propositional interpolation, which gives users some control over the process of interpolant generation. However, some of the leaves in the proof are theory clauses. For the propositional interpolation procedure to work, theory-specific interpolation procedures, *theory interpolators*, first compute interpolants for these theory lemmas. The interpolants for theory lemmas are used to annotate the corresponding leaves with partial interpolants, and the LIS interpolation procedure then works as usual. OpenSMT implements specific theory interpolators for the theory of equality and uninterpreted functions (EUF) and linear real arithmetic (LRA) [4]. EUF-interpolation system implements the standard interpolation computation from coloured congruence graphs [99, 156]. LRA-interpolation system is based on the standard idea of computing interpolants from proof of unsatisfiability based on the Farkas' lemma [156]. However, it does not require the detailed proof. Instead, it only needs the Farkas coefficients that witness the unsatisfiability of a system of linear inequalities.[3] Then, the interpolant is computed as the weighted sum of the A-part of the system of inequalities. For linear integer arithmetic (LIA), more advanced solving techniques are disabled when interpolation is required. Thus, the only theory lemmas that appear in the proof are either LRA-lemmas or split lemmas of the form $x \leq c \vee x \geq c + 1$ where x is a variable, and c is an integer constant. For the former, LRA-interpolator is used, and for the latter, OpenSMT implements a trick that treats such a clause as an input clause from the partition of the variable x.

2.4 Constrained Horn Clauses

Our presentation of constrained Horn clauses (CHC) is based on presentations in the existing literature [105, 176]. Consider a first-order theory $\mathcal{T}$ and a set $\mathcal{R}$ of uninterpreted predicates of fixed arity disjoint from the signature of $\mathcal{T}$. Then a constrained Horn clause is a formula

$$\varphi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n \implies H$$

where

- φ is a (interpreted) formula in the language of $\mathcal{T}$,
- each B_i is an application of a relation symbol $p \in \mathcal{R}$ to terms of $\mathcal{T}$,
- H is an application of a relation symbol $p \in \mathcal{R}$ to terms of $\mathcal{T}$, or *false*.
- All variables in the formula are implicitly universally quantified.

[3]The weighted sum of the system using Farkas coefficients results in a contradictory inequality $1 \leq 0$.

The antecedent of the implication is commonly denoted as the *body* and the consequent as the *head*. φ is referred to as the *constraint*. A clause with head equal to *false* is commonly called a *query*. A clause with *no* uninterpreted predicate in the body is called a *fact*.

Given a set of constrained Horn clauses HC over the uninterpreted predicates $\mathcal{R}$ and theory $\mathcal{T}$, we say that HC is satisfiable if there exists a model $\mathcal{M}$ of $\mathcal{T}$ extended with an interpretation for all the uninterpreted predicates $\mathcal{R}$, such that all the clauses are *valid* in $\mathcal{M}$. This is called *semantic* solvability in [176]. Often, we are interested not only in deciding the satisfiability, but also in obtaining the solution. This means we want to express the satisfying interpretation of predicates in the language of the theory $\mathcal{T}$, i.e., we want a mapping of the predicates to the set of formulas in the language of $\mathcal{T}$, $I : \mathcal{R} \to \mathit{FLA}$, such that each clause from HC is valid in $\mathcal{T}$ after the uninterpreted predicates are replaced by their interpretations. This is called *syntactic* solvability in [176]. Note that every system that is syntactically solvable is also semantically solvable, but not the other way around. The prominent example of a system that is semantically solvable, but not semantically solvable is the system that defines multiplication in Presburger arithmetic.

Example 2.2 ([176]). *The following set of Horn clauses is semantically solvable, but not syntactically solvable in Presburger arithmetic. The unique solution is the multiplication relation, which is not definable in Presburger arithmetic.*

$$\begin{aligned}
X = 0 \land Z = 0 &\implies \mathit{multA}(X, Y, Z) \\
\mathit{multA}(X1, Y, Z1) \land X = X1 + 1 \land Z = Z1 + Y &\implies \mathit{multA}(X, Y, Z) \\
X = 0 \land Z = 0 &\implies \mathit{multB}(X, Y, Z) \\
\mathit{multB}(X1, Y, Z1) \land X = X1 + 1 \land Z = Z1 + Y &\implies \mathit{multB}(X, Y, Z) \\
\mathit{multA}(X, Y, Z1) \land \mathit{multB}(X, Y, Z2) \land Z1 \neq Z2 &\implies \mathit{false}
\end{aligned}$$

Beside the question of solvability, it is often required to produce also a *certificate (witness)* for the answer. If a set of constrained Horn clauses is satisfiable, then the interpretation of the predicates is the certificate. On the other hand, the unsatisfiability of the clauses can be witnessed by a *ground derivation of false*, using resolution.

There are various methods for solving the CHC satisfiability problem. These include, e.g., machine-learning-based approaches (HoIce [52]), syntax-based approaches (FreqHorn [92, 93]), or automata-based approaches (Ultimate TreeAutomizer [75]). However, the most successful approaches currently seem to be the generalizations of classical model-checking techniques from software verification. These treat the CHC satisfiability problem as a reachability problem where the query in the CHC system defines the error states. The most successful and general CHC solvers Spacer [137] and Eldarica [117] fall into this category. It is interesting that in this reachability paradigm, the certificate of the system's satisfiability can be equivalently viewed as a certificate for *unreachability* of the error states defined by query, and similarly, the certificate of the system's unsatisfiability can be viewed as instructions how to *reach* an error state.

2.5 Transition Systems

Transition systems are a standard tool in computer science for modelling changing (evolving) systems. In particular, they can represent programs, where the states of a transition system are defined by possible program states (represented by, among others, values of program variables) and the instructions of the program define the possible transitions between the states. There exist different flavors of transition systems [163]. In our work, we consider *symbolic* representation of transition systems.

A transition system has a fixed set of typed variables X called *state* variables. Let X' denote the set of primed versions of variables from X, i.e., $X' = \{x' \mid x \in X\}$. These are commonly referred to as *next-state* variables. A *state formula* $F(X)$ is any quantifier-free formula over variables from X. A *transition formula* $Tr(X, X')$ is any quantifier-free formula over variables from $X \cup X'$. A *transition system* $\mathcal{S}$ (over X) is a pair $\langle Init, Tr\rangle$, where *Init* is a state formula denoting the initial states of the system and *Tr* is a transition formula that defines the *transition relation* of the system. A *state* s is a type-consistent assignment of variables from X, i.e., $s(x) \in Dom(x)$ for all $x \in X$. The transition relation *Tr* defines possible executions (transitions) of the system, i.e., how the system can change state. A state s can transition to a state s' iff $s, s' \models Tr$, i.e., *Tr* evaluates to true when the state variables X take the values assigned by s and the next-state variables X' take the values assigned by s'. A sequence of states $\langle s_0, s_1, \ldots, s_k\rangle$ is called a *trace* if $s_{i-1}, s_i \models Tr(X, X')$ for all $1 \leq i \leq k$. A state s is *k-reachable* in $\mathcal{S}$ (reachable in k steps) if there exists a trace $\langle s_0, s_1, \ldots, s_k\rangle$ such that $s_0 \vDash Init$ and $s_k = s$. A state is *reachable* if it is k-reachable for some finite k.

A state formula $F(X)$ holds in a state s if it evaluates to true under s and we write $s \vDash F$. The states s such that $s \vDash F$ are called the *F-states*. State formulas are often identified with the set of states where they hold and we freely move between these two representations. A state formula F is a *k-invariant* of the system if it holds in all states reachable in k or less steps. If F is a k-invariant then $\neg F$ is not reachable in k steps or less and we say that $\neg F$ is *k-inconsistent* with $\mathcal{S}$. When a concrete k is not important or not determined, or when we refer to multiple k-invariants but with different values of k, we use a more general term *bounded invariant*. A bounded invariant F is thus a state formula for which there exists k such that F is a k-invariant. An *invariant* is a state formula that is a k-invariant for all k, i.e., it holds in *all* reachable states of $\mathcal{S}$.

Similar to state formulas, transition formulas are identified with binary relations over the set of states. For example, the identity relation $Id(X, X')$ corresponds to the transition formula $\bigwedge_{x \in X} x = x'$.

In this thesis, we consider verification of *safety* properties of transition systems. Given a transition system $\mathcal{S}$ and a state formula P, the goal of verification is to prove that P is valid on all reachable states of $\mathcal{S}$, or equivalently that $\neg P$ is not reachable. We say that the system is *safe* with respect to P if P is indeed an invariant of the system, and we say that it is *unsafe* if there exists a finite trace starting from an initial state and ending in a $\neg P$-state. We often refer to $\neg P$-states as *error* states and denote them as *Error*, i.e., $Error \equiv \neg P$.

```
assume(x <= 0);
while (x < 5) {
  x = x + 1;
}
assert(x < 10);
```

$$\begin{aligned} \mathit{Init} &\equiv x \leq 0 \\ \mathit{Tr} &\equiv x < 5 \land x' = x + 1 \\ \mathit{Error} &\equiv x \geq 5 \land \neg(x < 10) \end{aligned}$$

$$\begin{aligned} x \leq 0 &\implies \mathit{Inv}(x) \\ \mathit{Inv}(x) \land x < 5 \land x' = x + 1 &\implies \mathit{Inv}(x') \\ \mathit{Inv}(x) \land x \geq 5 \land \neg(x < 10) &\implies \mathit{false} \end{aligned}$$

Figure 2.3. An example program with a loop, the corresponding transition system, and its translation to CHC

2.5.1 Safety Verification of Transition Systems as CHC Satisfiability

In the context of transition systems, a safety verification problem is defined by a triple $\langle \mathit{Init}, \mathit{Tr}, \mathit{Error} \rangle$ where *Init* is a state formula defining the initial states of the system, *Tr* is a transition formula defining the transition relation and *Error* is a state formula defining the error states. If an error state is reachable, i.e., there exists a trace starting in an initial state and ending in an error state, then system is unsafe. If no such trace exists, it is safe. This can be easily modeled in the CHC framework with a single uninterpreted predicate *Inv* and the following three clauses.

$$\begin{aligned} \mathit{Init}(X) &\implies \mathit{Inv}(X) \\ \mathit{Inv}(X) \land \mathit{Tr}(X, X') &\implies \mathit{Inv}(X') \\ \mathit{Inv}(X) \land \mathit{Error}(X) &\implies \mathit{false} \end{aligned}$$

The solution to this system, i.e., the interpretation of *Inv* that makes all clauses valid, is a *safe inductive invariant* of the transition system. The first clause says that the invariant must hold in all initial states. The second clause says that the invariant is inductive, i.e., it is closed under the transition relation of the system. The third clause says that the invariant is safe, i.e., it is disjoint with the error states. Such safe inductive invariant is a witness that no error state can be reached in the system. On the other hand, every proof of unsatisfiability defines a trace of the system from some initial to some error state.

To illustrate the CHC modelling and its application in software verification, consider the example program from [105] in Figure 2.3. This is a simple loop with a safety property expressed as an assertion. The problem of program safety can be translated to a safety verification problem of a transition system. One way to prove safety is to find a safe inductive invariant. This is modelled as a system of Horn clauses. In this case, the system of Horn clauses is satisfiable with a solution $\mathit{Inv}(x) \equiv x \leq 5$. It is easy to verify that this is indeed a safe inductive invariant of the transition system and consequently of our example loop.

Chapter 3

Decomposing Farkas Interpolants

Craig interpolants play a crucial role in many modern verification techniques. Interpolation-based model-checking algorithms typically rely on an interpolating SMT solver to generate interpolants from unsatisfiable queries. Internally, SMT solvers produce interpolants by structural induction over the proof of unsatisfiability. Interpolation procedures build the interpolant while traversing the proof from leaf nodes to the root. In SMT, many of the leaf nodes represent a theory conflict—a conjunction of theory literals that is unsatisfiable in that theory. Proof-based interpolation procedures thus rely on subprocedures that compute interpolants for the theory conflicts in the leaf nodes. Any theory that contains integer or real linear arithmetic will produce theory conflicts that represent an unsatisfiable system of linear inequalities. Given such a system partitioned into two parts (A and B), the main way to compute an interpolant is to utilize Farkas coefficients that witness the unsatisfiability of the system, based on the Farkas' lemma [90]. In modern SMT solvers, Farkas coefficients are computed as a side product in the Simplex-based algorithm of Dutertre and de Moura [84]. An interpolant can then be obtained by summing up the A contribution to the conflict. We refer to an interpolant computed in this way as *Farkas interpolant.* Farkas interpolant always takes the form of a *single* inequality. In an application such as model checking, this can be both an advantage or disadvantage: In some cases, the inequality introduces a new relation between variables into the abstraction in the model-checking algorithm. Taking this relation into account then leads the model checker to discover a key part of a safe inductive invariant. In other cases however, the single inequality is too weak and is the source of divergence of the model checker.

In this chapter, we study Farkas interpolants and show that under some conditions, it is possible to *decompose* the interpolant into a *conjunction* of inequalities that logically implies the Farkas interpolant. Farkas interpolation summarizes the contribution of the A-part of the conflict into a single inequality, losing information about finer constraints on subsets of variables in the process. On the other hand, full quantifier elimination preserves as much information about the variables from the A-part as possible; however, this might be very expensive. Our approach represents a sweet spot between these two extremes. The decomposition of the weighted sum of the A-part can still be computed efficiently and, if possible, preserves much more information about variables from the A-part than Farkas interpolation.

3.1 Preliminaries

3.1.1 Linear Arithmetic and Linear Algebra

We use the letters x, y, z to denote variables and c, k to denote constants. Vector of n variables is denoted by $\mathbf{x} = (x_1, \ldots, x_n)^\intercal$ where n is usually known from context. $\mathbf{x}[i]$ denotes the element of $\mathbf{x}$ at position i, i.e. $\mathbf{x}[i] = x_i$. The vector of all zeroes is denoted as $\mathbf{0}$, and $\mathbf{e_i}$ denotes the unit vector with $\mathbf{e_i}[i] = 1$ and $\mathbf{e_i}[j] = 0$ for $j \neq i$. For two vectors $\mathbf{x} = (x_1, \ldots, x_n)^\intercal$ and $\mathbf{y} = (y_1, \ldots, y_n)^\intercal$ we say that $\mathbf{x} \leq \mathbf{y}$ iff $x_i \leq y_i$ for each $i \in \{1, \ldots, n\}$. $\mathbb{Q}$ denotes the set of rational numbers, $\mathbb{Q}^n$ the n-dimensional vector space of rational numbers and $\mathbb{Q}^{m \times n}$ the set of rational matrices with m rows and n columns. A transpose of matrix M is denoted as $M^\intercal$. A *kernel* (or *nullspace*) of a matrix M is the vector space $ker(M) = \{\mathbf{x} \mid M\mathbf{x} = \mathbf{0}\}$. A matrix is said to be in *Row Echelon Form* (REF) if all non-zero rows are above all rows containing only zeros and the leading coefficient (first non-zero value) of each row is always strictly to the right of the leading coefficient of the row above. A matrix is said to be in *Reduced Row Echelon Form* (RREF) if it is in REF, the leading entry of each non-zero row is 1, and each column containing the leading entry of some row has zeros everywhere else. REF of a matrix can be obtained by Gaussian elimination, while RREF can be obtained by Gauss-Jordan elimination.

We adopt the notation of matrix product for linear arithmetic. For a linear term $l = c_1x_1 + \cdots + c_nx_n$, we write $\mathbf{c}^\intercal\mathbf{x}$ to denote l. Without loss of generality we assume that all linear inequalities are of the form $\mathbf{c}^\intercal\mathbf{x} \bowtie c$ with $\bowtie \in \{\leq, <\}$. By linear system over variables $\mathbf{x}$ we mean a finite set of linear inequalities $S = \{C_i \mid 1 \leq i \leq m\}$, where each C_i is a linear inequality over $\mathbf{x}$. Note that from the logical perspective, each C_i is an atom in the language of the theory of linear arithmetic; thus system S can be expressed as a formula $\bigwedge_{i=1}^m C_i$ and we use these representations interchangeably. A linear system is satisfiable if there exists an evaluation of variables that satisfies all inequalities; otherwise, it is unsatisfiable. This is the same as the (un)satisfiability of the formula representing the system.

We extend the matrix notation also to the whole linear system. For the sake of simplicity we use $\leq$ instead of $\bowtie$, even if the system contains a mix of strict and non-strict inequalities. The only important difference is that a (weighted) sum of a linear system (as defined below) results in a strict inequality, instead of a non-strict one, when at least one strict inequality is present in the sum with a non-zero coefficient. The theory, proofs and algorithm remain valid also in the presence of strict inequalities. We write $C\mathbf{x} \leq \mathbf{c}$ to denote the linear system S where C denotes the matrix of all coefficients of the system, $\mathbf{x}$ are the variables and $\mathbf{c}$ is the vector of the right sides of the inequalities. With the matrix notation, we can easily express the sum of (multiples) of inequalities. Given a system of inequalities $C\mathbf{x} \leq \mathbf{c}$ and a vector of "weights" (multiples) of the inequalities $\mathbf{k} \geq \mathbf{0}$, the inequality that is the (weighted) sum of the system can be expressed as $\mathbf{k}^\intercal C\mathbf{x} \leq \mathbf{k}^\intercal\mathbf{c}$.

3.1.2 Craig Interpolation Based on Farkas' Lemma

In linear arithmetic, the interpolation problem is an unsatisfiable linear system S of linear inequalities partitioned into two parts: A and B. Modern SMT solvers typically compute interpolant for such a system based on Farkas' lemma [90, 181]. Farkas' lemma states that for an unsatisfiable system of linear inequalities $S \equiv C\mathbf{x} \leq \mathbf{c}$ there exist *Farkas coefficients* $\mathbf{k} \geq \mathbf{0}$ such that $\mathbf{k}^\intercal C\mathbf{x} \leq \mathbf{k}^\intercal\mathbf{c} \equiv 0 \leq -1$. In other words, the weighted sum of the system given by the Farkas coefficients is a contradictory inequality. If a strict inequality is part of the sum, the result might also be $0 < 0$.

The idea behind the interpolation algorithm based on Farkas coefficients is simple. Intuitively, given the partitioning of the linear system into A and B, we compute only the weighted sum of A. It is not hard to see that this sum is an interpolant. It follows from A because a weighted sum of a linear system with non-negative weights is always implied by the system. It is inconsistent with B because its sum with the weighted sum of B (using Farkas coefficients) is a contradictory inequality by Farkas' lemma. Finally, it cannot contain any A-local variables, as can be seen from the following reasoning: All variables are eliminated in the weighted sum of the whole system. Since A-local variables are by definition absent in B, they must be eliminated already in the weighted sum of A.

More formally, for an unsatisfiable linear system $S := C\mathbf{x} \leq \mathbf{c}$ over n variables, where $C \in \mathbb{Q}^{m\times n}, \mathbf{c} \in \mathbb{Q}^m$, and its partition to $A := C_A\mathbf{x} \leq \mathbf{c_A}$ and $B := C_B\mathbf{x} \leq \mathbf{c_B}$, where $C_A \in \mathbb{Q}^{k\times n}$, $C_B \in \mathbb{Q}^{l\times n}$, $\mathbf{c_A} \in \mathbb{Q}^k$, $\mathbf{c_B} \in \mathbb{Q}^l$ and $k + l = m$, there exist Farkas coefficients $\mathbf{k}^\intercal = (\mathbf{k}_\mathbf{A}^\intercal\ \mathbf{k}_\mathbf{B}^\intercal)$ such that

$$(\mathbf{k}_\mathbf{A}^\intercal\ \mathbf{k}_\mathbf{B}^\intercal)\begin{pmatrix} C_A \\ C_B \end{pmatrix} = 0, (\mathbf{k}_\mathbf{A}^\intercal\ \mathbf{k}_\mathbf{B}^\intercal)\begin{pmatrix} \mathbf{c_A} \\ \mathbf{c_B} \end{pmatrix} = -1,$$

and the *Farkas interpolant* for (A, B) is the inequality

$$I^F := \mathbf{k}_\mathbf{A}^\intercal C_A\mathbf{x} \leq \mathbf{k}_\mathbf{A}^\intercal\mathbf{c_A}. \tag{3.1}$$

3.2 Motivation

To motivate our work, consider the code in Figure 3.1.[1] In this code, the '$*$' character represents a non-deterministic choice (e.g., user input); thus, the body of the while loop can be executed any number of times. The assert statement captures the property of the program that variable 'x' should always be non-negative after exiting the while loop.

This code can be modelled as a transition system $S = (I, T, Err)$ given in Equation (3.2); here, I and Err are predicates that capture the initial and error states,

[1] This example is commonly used in the literature [42, 179].

```
x = 0;
y = 0;
while (*) {
   x = x + y;
   y = y + 1;
}
assert(x >= 0);
```

Figure 3.1. Motivating example

respectively, and T is the transition relation. The symbols x, y are real variables, and x', y' are their next-state versions.

$$S = \begin{cases} I := (x = 0) \wedge (y = 0), \\ T := (x' = x + y) \wedge (y' = y + 1), \\ Err := (x < 0) \end{cases} \tag{3.2}$$

The aforementioned example is one variant from a family of similar transition systems that are known not to converge in straightforward implementations of IC3-based algorithms using LRA interpolation. To prove the safety of the transition system (I, T, Err) we search for a safe inductive invariant, i.e., a predicate R that satisfies (1) $I(X) \to R(X)$, (2) $R(X) \wedge T(X, X') \to R(X')$, and (3) $R(X) \wedge Err(X) \to \bot$.

We demonstrate the problem that occurs in model checking when using Farkas interpolants on a simplified run of a model checker for our example. After checking that the initial state satisfies the property $P := x \geq 0$ (the negation of Err), the inductiveness of P is checked. The inductive check is reduced to a satisfiability check of a formula representing the question whether it is possible to reach a $\neg P$-state (a state where $\neg P$ holds) by one step from any P-state:

$$x \geq 0 \wedge x' = x + y \wedge y' = y + 1 \wedge x' < 0.$$

This formula is satisfiable, and a generalized counterexample to induction (CTI) is extracted. In our case, the CTI is $x + y < 0$.[2] This means that if we make one step from a P-state that additionally satisfies $x + y < 0$ we end up in a $\neg P$-state. Therefore, we have to check if this CTI is consistent with the initial states. This is again encoded as a satisfiability check of a formula

$$x = 0 \wedge y = 0 \wedge x + y < 0.$$

This formula is unsatisfiable, and we can extract an *interpolant* to obtain a generalized reason why this CTI is not consistent with the initial states (not reachable in 0 steps in our

[2]The exact procedure for obtaining the CTI is not important for the current discussion.

system). The interpolant is computed for the partitioning $(x = 0 \wedge y = 0, x + y < 0)$. The Farkas interpolant for this partitioning is $x + y \geq 0$, and we denote it as L_1. Interpolation properties guarantee that L_1 is valid in all initial states. Moreover, P is inductive *relative to* L_1, formally

$$x \geq 0 \wedge x + y \geq 0 \wedge x' = x + y \wedge y' = y + 1 \implies x' \geq 0.$$

This means that by making one step from a P-state that is also an L_1-state we always end up in a P-state again. However, now we need to show that L_1 holds in all reachable states. We check if L_1 is inductive (even relative to P). Similarly as before, we encode this as a satisfiability check of a formula

$$x + y \geq 0 \wedge x \geq 0 \wedge x' = x + y \wedge y' = y + 1 \wedge x' + y' < 0.$$

Again, this formula is satisfiable, and a generalized CTI is $x + 2y < -1$. This CTI is refuted as inconsistent with the initial states similarly to the first one. The formula

$$x = 0 \wedge y = 0 \wedge x + 2y < -1$$

is unsatisfiable and Farkas interpolant generalizing the refutation is $L_2 := x + 2y \geq 0$. Similarly as before, it can be easily checked that L_1 is inductive *relative* to L_2, but L_2 is not inductive (not even relative to P and L_1). The CTI is $x + 3y < -1$, it is refuted by a Farkas interpolant $L_3 := x + 3y \geq 0$. L_2 is now inductive relative to L_3, but L_3 is not inductive, etc. The model checker diverges, since for L_n a CTI $x + ny < -1$ is discovered and a new obligation to show inductiveness of L_{n+1} is generated.

However, let us get back to the first interpolation query $(x = 0 \wedge y = 0, x + y < 0)$. Farkas interpolation, which always computes an interpolant in the form of a *single* inequality, is not the only option. It is possible to compute an interpolant that is a *conjunction* of inequalities. In our case, $L := x \geq 0 \wedge y \geq 0$ is also an interpolant. This interpolant L is stronger than the Farkas interpolant; the property P is inductive relative to L, and, most importantly, L is inductive:

$$\begin{aligned}(x \geq 0 \wedge y \geq 0) \wedge x' = x + y \wedge& \\ \wedge y' = y + 1 &\implies (x' \geq 0 \wedge y' \geq 0)\end{aligned}$$

is a valid formula. Actually, P follows from L, so L represents the inductive strengthening of P that witnesses the safety of our system.

In this work, we present an approach that allows the computation of Craig interpolants in LRA in this conjunctive form.

3.3 Decomposed Interpolants

In this section, we present our new approach to computing interpolants in linear arithmetic based on Farkas coefficients. The definition of Farkas interpolant in Equation (3.1) corresponds to the weighted sum of A-part of the unsatisfiable linear system. This sum can be decomposed into j sums by decomposing the vector $\mathbf{k_A}$ into j vectors

$$\mathbf{k_A} = \sum_{i=1}^{j} \mathbf{k_{A,i}}, \text{ with } \mathbf{0} \leq \mathbf{k_{A,i}} \leq \mathbf{k_A} \text{ for all } i, \tag{3.3}$$

thus obtaining j inequalities

$$I_i := \mathbf{k}_{\mathbf{A,i}}^{\mathsf{T}} C_A \mathbf{x} \leq \mathbf{k}_{\mathbf{A,i}}^{\mathsf{T}} \mathbf{c_A} \tag{3.4}$$

If $\mathbf{k_{A,i}}$ are such that the left-hand side of the inequalities I_i contains only shared variables, the decomposition has an interesting application in interpolation, as illustrated below.

Definition 3.1 (decomposed interpolants)**.** *Given an interpolation instance (A, B), if there exists a sum from Equation* (3.3) *such that the left side of Equation* (3.4) *contains only shared variables for all $1 \leq i \leq j$, then the set of inequalities $D = \{I_1, \ldots, I_j\}$ is a* decomposition. *In that case the formula $\bigwedge_{i=1}^{j} I_i$ is a* decomposed interpolant *(DI) of size j for (A, B).*

The decomposed interpolants are proper interpolants, as stated in the following theorem.

Theorem 3.2. *Let (A, B) be an interpolation problem in linear arithmetic. If $D = \{I_1, \ldots, I_k\}$ is a decomposition, then $I^D = I_1 \wedge \ldots \wedge I_k$ is an interpolant for (A, B).*

Proof. Let $I^D = I_1 \wedge \ldots \wedge I_k$. First, $A \implies I^D$ since for all I_i, $A \implies I_i$. This is immediate from the fact that A is a system of linear inequalities $C_A\mathbf{x} \leq \mathbf{c_A}$, $I_i = (\mathbf{k}_{\mathbf{A,i}}^{\mathsf{T}} C_A \mathbf{x} \leq \mathbf{k}_{\mathbf{A,i}}^{\mathsf{T}} \mathbf{c_A})$ and $\mathbf{0} \leq \mathbf{k_{A,i}}$.

Second, $I^D \wedge B \implies \bot$ since I^D implies Farkas interpolant I^F. This holds because $\mathbf{k_A} = \sum_i \mathbf{k_{A,i}}$ and $\mathbf{0} \leq \mathbf{k_{A,i}}$.

Third, I^D contains only the shared variables by the definition of decomposition (Definition 3.1). Therefore, I^D is an interpolant. □

Each interpolation instance has a *DI* of size one, a *trivial* decomposition, corresponding to the Farkas interpolant of Equation (3.1). However, interpolation problems, in general, can admit bigger decompositions. In the following, we give a concrete example of an instance with decomposition of size two.

Example 3.3. *Let (A, B) be an interpolation problem in linear arithmetic with $A = (x_1 + x_2 \leq 0) \wedge (x_1 + x_3 \leq 0) \wedge (-x_1 \leq 0)$ and $B = (-x_2 - x_3 \leq -1)$. The linear systems corresponding to A and B are*

$$C_A = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ -1 & 0 & 0 \end{pmatrix}, \quad \mathbf{c_A} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$C_B = \begin{pmatrix} 0 & -1 & -1 \end{pmatrix}, \quad \mathbf{c_B} = \begin{pmatrix} -1 \end{pmatrix}.$$

Farkas coefficients are

$$\mathbf{k}_\mathbf{A}^\intercal = \begin{pmatrix}1 & 1 & 2\end{pmatrix} \text{ and } \mathbf{k}_\mathbf{B}^\intercal = \begin{pmatrix}1\end{pmatrix},$$

while Farkas interpolant for (A, B) is the inequality $I^F := x_2 + x_3 \leq 0$. However, if we decompose $\mathbf{k_A}$ into

$$\mathbf{k}_\mathbf{A,1}^\intercal = \begin{pmatrix}1 & 0 & 1\end{pmatrix} \text{ and } \mathbf{k}_\mathbf{A,2}^\intercal = \begin{pmatrix}0 & 1 & 1\end{pmatrix},$$

we obtain the decomposition $\{x_2 \leq 0, x_3 \leq 0\}$ producing the decomposed interpolant $I^{DI} := x_2 \leq 0 \wedge x_3 \leq 0$ of size two.

3.3.1 Strength-Based Ordering of Decompositions

Decomposition of Farkas coefficients for a single interpolation problem is in general not unique. However, we can provide some structure to the space of possible interpolants by ordering interpolants with respect to their logical strength. To achieve this, we define the *coarseness* of a decomposition based on its ability to partition the terms of the interpolant into finer sums, and then prove that coarseness provides us with a way of measuring the interpolant strength.

Definition 3.4. *Let D_1, D_2 denote two decompositions of the same interpolation problem of size m, n, respectively, where $n < m$. Let $(\mathbf{q_1}, \ldots, \mathbf{q_m})$ denote the decomposition of Farkas coefficients corresponding to D_1 and let $(\mathbf{r_1}, \ldots, \mathbf{r_n})$ denote the decomposition of Farkas coefficients corresponding to D_2. We say that decomposition D_1 is* finer *than D_2 (or equivalently D_2 is* coarser *than D_1) and denote this as $D_1 \prec D_2$ when there exists a partitioning $P = \{p_1, \ldots, p_n\}$ of the set $\{\mathbf{q_1}, \ldots, \mathbf{q_m}\}$ such that for each i with $1 \leq i \leq n$, $\mathbf{r_i} = \sum_{\mathbf{q} \in p_i} \mathbf{q}$.*

Interpolants of decompositions ordered by their coarseness can be ordered by logical strength, as stated by the following lemma:

Lemma 3.5. *Assume D_1, D_2 are two decompositions of the same interpolation problem such that $D_1 \prec D_2$. Let I^{D_1}, I^{D_2} be the decomposed interpolants corresponding to D_1, D_2. Then I^{D_1} implies I^{D_2}.*

Proof. Informally, the implication follows from the fact that each linear inequality of I^{D_2} is a sum of some inequalities in I^{D_1}.

Formally, let I_i denote the i-th inequality in I^{D_2}. Then $I_i = (\mathbf{r_i}^\intercal C_A \mathbf{x} \leq \mathbf{r_i}^\intercal \mathbf{c_A})$. Since $D_1 \prec D_2$, there is a set $\{I_{i_1}, \ldots, I_{i_j}\} \subseteq D_1$ such that for each k with $1 \leq k \leq j$, $I_{i_k} = (\mathbf{q}_{\mathbf{i_k}}^\intercal C_A \mathbf{x} \leq \mathbf{q}_{\mathbf{i_k}}^\intercal \mathbf{c_A})$ and $\mathbf{r_i} = \sum_{k=1}^{j} \mathbf{q}_{\mathbf{i_k}}$.

Since $\mathbf{q}_{\mathbf{i_k}} \geq \mathbf{0}$, it holds that $I_{i_1} \wedge \cdots \wedge I_{i_j} \implies I_i$. This means that I^{D_1} implies every conjunct of I^{D_2}. □

Note that the trivial, single-element decomposition corresponding to Farkas interpolant is the greatest element of this decomposition ordering. Also, for any decomposition of size more than one, replacing any number of elements by their sum yields a coarser decomposition.

Finally, we emphasize that it is difficult to argue about the suitability of a decomposition for a particular purpose based solely on strength. For example, a user may opt for a coarser decomposition because summing up just some elements of a decomposition may result in eliminating a shared variable.

3.3.2 Strength of Dual Interpolants

Before we describe the details of the decomposing interpolation procedure, we extend the picture of interpolation strength related to the decomposed interpolants.

Some applications of interpolation can benefit from computing coarser over-approximation (i.e., weaker interpolants). For example, a weaker function summary can cover more changes in an upgrade checking scenario [182], and weaker over-approximations of reachability in a transition system can converge to fix-point faster [81]. Using the notion of *dual interpolation*, decompositions can also be used to compute interpolants *weaker* than Farkas interpolant (or even its dual).

Given an interpolation problem (A, B) and an interpolation procedure *Itp*, we denote the interpolant computed by *Itp* for (A, B) as $\mathit{Itp}(A, B)$. Then Itp' denotes the *dual* interpolation procedure, which works as follows: $\mathit{Itp}'(A, B) = \neg \mathit{Itp}(B, A)$. The well-known duality theorem for interpolation states that Itp' is a correct interpolation procedure.

Let us denote the interpolation procedure based on Farkas' lemma as Itp_F and the decomposing interpolation procedure as Itp_{DI}. The relation between Itp_F and its dual Itp'_F has been established in [8], namely that $\mathit{Itp}_F(A, B) \implies \mathit{Itp}'_F(A, B)$. We have shown in Lemma 3.5 that a decomposed interpolant always implies Farkas interpolant computed from the same Farkas coefficients. Formally, $\mathit{Itp}_{DI}(A, B) \implies \mathit{Itp}_F(A, B)$. Similar result can be established for the dual interpolation procedures: As $\mathit{Itp}_{DI}(B, A) \implies \mathit{Itp}_F(B, A)$, it follows that $\neg \mathit{Itp}_F(B, A) \implies \neg \mathit{Itp}_{DI}(B, A)$ and consequently $\mathit{Itp}'_F(A, B) \implies \mathit{Itp}'_{DI}(A, B)$.

Combining the results on logical strength together we obtain a chain of implications

$$\mathit{Itp}_{DI}(A, B) \implies \mathit{Itp}_F(A, B) \implies \mathit{Itp}'_F(A, B) \implies \mathit{Itp}'_{DI}(A, B).$$

Note that while both Itp_F and Itp'_F compute interpolants as a single inequality and interpolants produced by Itp_{DI} are *conjunctions* of inequalities, interpolants produced by Itp'_{DI} are *disjunctions* of inequalities.

In the following section, we describe the details of the Itp_{DI} interpolation procedure.

3.4 Finding Decompositions

In this section, we present our approach for finding decompositions for linear arithmetic interpolation problems given their Farkas coefficients.

We focus on the task of finding decomposition of $\mathbf{k_A^\intercal} C_A \mathbf{x}$. Recall that $C_A \in \mathbb{Q}^{l \times n}$ and $\mathbf{x}$ is a vector of variables of length n. Without loss of generality assume that there are no B-local variables since columns of C_A corresponding to B-local variables would contain all zeroes by definition in any case.

Furthermore, without loss of generality, assume the variables in the inequalities of A are ordered such that all A-local variables are before the shared ones. Then let us write

$$C_A = \begin{pmatrix} L & S \end{pmatrix}, \quad \mathbf{x}^\intercal = \begin{pmatrix} \mathbf{x}_L{}^\intercal & \mathbf{x}_S{}^\intercal \end{pmatrix} \tag{3.5}$$

with $\mathbf{x}_L$ the vector of A-local variables of size p, $\mathbf{x}_S$ the vector of shared variables of size q, $n = p + q$, $L \in \mathbb{Q}^{l \times p}$ and $S \in \mathbb{Q}^{l \times q}$. We know that $\mathbf{k_A^\intercal} L = \mathbf{0}$ and the goal is to find $\mathbf{k_{A,i}}$ such that $\sum_i \mathbf{k_{A,i}} = \mathbf{k_A}$ and for each i $\mathbf{0} \leq \mathbf{k_{A,i}} \leq \mathbf{k_A}$ and $\mathbf{k_{A,i}^\intercal} L = \mathbf{0}$.

In the following, we will consider two cases for computing the decompositions. We first study a common special case where system A contains rows with no local variables, and give a linear-time algorithm for computing the decompositions. We then move to the general case where the rows of A contain local variables and provide a decomposition algorithm based on computing a vector basis for a null space of a matrix obtained from A.

3.4.1 Trivial Elements

First, consider a situation where there is a linear inequality with no local variables. This means there is a row j in C_A (denoted as C_{Aj}) such that all entries in columns corresponding to local variables are 0, i.e., $L_j = \mathbf{0}^\intercal$. Then $\{I_1, I_2\}$ for $\mathbf{k_{A,1}} = \mathbf{k_A}[j] \times \mathbf{e_j}$ and $\mathbf{k_{A,2}} = \mathbf{k_A} - \mathbf{k_{A,1}}$ is a decomposition. Intuitively, any linear inequality that contains only shared variables can form a stand-alone element of a decomposition. When looking for finest decomposition, we do this iteratively for all inequalities with no local variables. In the next part, we show how to look for a non-trivial decomposition when dealing with local variables.

3.4.2 Decomposing in the Presence of Local Variables

For this section, assume that L has no zero rows (we have shown above how to deal with such rows). We are going to search for a non-trivial decomposition starting with the following observation:

Observation 3.6. $\mathbf{k_A^\intercal} L = 0$. *Equivalently, there are no A-local variables in the Farkas interpolant. It follows that* $L^\intercal \mathbf{k_A} = 0$ *and* $\mathbf{k_A}$ *is in the* kernel *of* $L^\intercal$.

Let us denote by $\mathbb{K} = ker(L^\intercal)$ the kernel of $L^\intercal$.

Theorem 3.7. *Let* $\mathbf{v_1}, \ldots, \mathbf{v_n}$ *be vectors from* $\mathbb{K}$ *such that* $\exists \alpha_1, \ldots, \alpha_n$ *with* $\alpha_i \mathbf{v_i} \geq \mathbf{0}$ *for all i and* $\mathbf{k_A} = \sum_{i=1}^n \alpha_i \mathbf{v_i}$. *Then* $\{\mathbf{w_1}, \ldots, \mathbf{w_n}\}$ *for* $\mathbf{w_i} = \alpha_i \mathbf{v_i}$ *is a decomposition of* $\mathbf{k_A}$ *and* $D = \{I_1, \ldots, I_n\}$ *for* $I_i := \mathbf{w_i} C_A \mathbf{x} \leq \mathbf{c_A}$ *is a decomposition, i.e., the formula* $I^D = \bigwedge_{i=1}^n I_i$ *is a decomposed interpolant.*

```
input  : matrix $M$, vector $\mathbf{v}$ such that $\mathbf{v} \in ker(M)$ and $\mathbf{v} > \mathbf{0}$
output : $\{\mathbf{w_1}, \ldots, \mathbf{w_n}\}$, a decomposition of $\mathbf{v}$, such that $\mathbf{w_i} \in ker(M), \mathbf{w_i} \geq \mathbf{0}$
         and $\sum \mathbf{w_i} = \mathbf{v}$
1  $M \leftarrow$ RREF($M$)
2  $n \leftarrow$ Nullity($M$)
3  if $n = 1$ then return $\{\mathbf{v}\}$
4  $(\mathbf{b_1}, \ldots, \mathbf{b_n}) \leftarrow$ KernelBasis($M$)
5  $(\alpha_1, \ldots, \alpha_n) \leftarrow$ Coordinates($\mathbf{v}, (\mathbf{b_1}, \ldots, \mathbf{b_n})$)
6  assert $\alpha_k > 0$ for each $k = 1, \ldots, n$
7  while $\exists i, j$ such that $\mathbf{b_i}_j < 0$ do
8      $C \leftarrow 1 + \frac{-\mathbf{b_i}_j \alpha_i}{\mathbf{v}_j}$
9      $\mathbf{b_i} \leftarrow \mathbf{b_i} + \frac{-\mathbf{b_i}_j}{\mathbf{v}_j} \mathbf{v}$
10     $(\alpha_1, \ldots, \alpha_n) \leftarrow (\frac{\alpha_1}{C}, \ldots, \frac{\alpha_n}{C})$
11     assert $\alpha_k > 0$ for each $k = 1, \ldots, n$
12     assert $\mathbf{v} = \sum_{k=1}^{n} \alpha_k \mathbf{b_k}$
13 assert $\mathbf{b_k} \geq \mathbf{0}$ for each $k = 1, \ldots, n$
14 return $\{\alpha_1 \mathbf{b_1}, \ldots, \alpha_n \mathbf{b_n}\}$
```

Algorithm 3.1. Algorithm for decomposition of Farkas coefficients

Proof. The theorem follows from the definition of decomposition (Definition 3.1). From the assumptions of the theorem, we immediately obtain $\mathbf{k_A} = \sum_{i=1}^{n} \mathbf{w_i}$ and $\mathbf{w_i} \geq \mathbf{0}$. Moreover, $\mathbf{w_i} \in \mathbb{K}$, since $\mathbf{v_i} \in \mathbb{K}$ and $\mathbf{w_i} = \alpha_i \mathbf{v_i}$. As a consequence, $L^\intercal \mathbf{w_i} = 0$ and it follows that there are no A-local variables in $\mathbf{w_i}^\intercal C_A \mathbf{x}$. □

Note that Theorem 3.7 permits redundant components of a decomposition. Consider vectors $\mathbf{w_1}, \mathbf{w_2}, \mathbf{w_3} \in \mathbb{K}$ that are part of a decomposition in the sense of Theorem 3.7 and that $\mathbf{w_3} = \mathbf{w_1} + \mathbf{w_2}$. Then $I_1 \wedge I_2 \implies I_3$ and I_3 is a *redundant conjunct* in the corresponding decomposed interpolant.

Good candidates that satisfy most of the assumptions of Theorem 3.7 (and avoid redundancies) are bases of the vector space $\mathbb{K}$. If $B = \{\mathbf{b_1}, \ldots, \mathbf{b_n}\}$ is a basis of $\mathbb{K}$ such that $\mathbf{k_A} = \sum_{i=1}^{n} \alpha_i \mathbf{b_i}$ with $\alpha_i \mathbf{b_i} \geq \mathbf{0}$ for all i, then $\{\alpha_1 \mathbf{b_1}, \ldots, \alpha_n \mathbf{b_n}\}$ is a decomposition. Our solution for computing the decomposition of Farkas coefficients $\mathbf{k_A}$ is described in Algorithm 3.1. It is based on the above idea of computing bases of $ker(L^\intercal)$. First, after transforming the matrix to the RREF form, we compute a basis of the kernel using the standard linear-algebra algorithm. The basis is almost what we want, except that some vectors of this basis can have negative coefficients. In such a case, our algorithm gradually updates the basis until all vectors from the basis are non-negative while preserving all the necessary properties. Such a basis is used to compute the desired decomposition. Now, we describe our algorithm in detail, show its termination and correctness, and discuss its complexity.

The algorithm runs on the matrix $M = L^\intercal$ and vector $\mathbf{v} = \mathbf{k_A}$. At the beginning, the *Reduced Row Echelon Form* (RREF) of the matrix is computed (recall definition of RREF from Section 3.1). Importantly, the transformation of a matrix to RREF preserves its kernel. The dimension of the kernel, known as *nullity*, can now be efficiently computed using *Rank-Nullity Theorem*, which states that the nullity of a matrix is equal to the number of its columns minus its rank. For a matrix in RREF, the rank is simply the number of non-zero rows.

We already know that there is a non-zero vector in the kernel; therefore the nullity of the matrix is at least one. If it is *exactly* one (line 3), then no non-trivial decomposition of the vector exists. Intuitively, this means that the Farkas coefficients represent *the unique way* (up to positive scalar multiples) of summing up the inequalities of A-part to eliminate the A-local variables. However, if the nullity is greater than one, it is possible to compute a decomposition of size equal to the nullity.

Initial basis computation. First, a basis of the kernel of the matrix in RREF is computed by a standard algorithm (see, e.g., [10]). This algorithm ensures that the coordinates of $\mathbf{v}$, with respect to the basis it computes, are positive (lines 5, 6). Since this is an important property, we include the description of the algorithm with the proof. Given a matrix M in RREF with m columns, each column is denoted as either pivot or non-pivot. A pivot column contains the first non-zero entry for a particular row, non-pivot column does not. We say that a non-pivot column is *free*. The number of free columns is exactly the nullity of the matrix, i.e., n, and the number of pivot columns is $m - n$. Due to the need to iterate over the pivot and free columns separately, we introduce additional notation: we use $f \in \{1, \ldots, n\}$ to iterate over the free columns, $p \in \{1, \ldots, m-n\}$ to iterate over the pivot columns, and we use mapping functions $F\colon \{1, \ldots, n\} \to \{1, \ldots, m\}$ and $P\colon \{1, \ldots, m-n\} \to \{1, \ldots, m\}$ to get the original column indices in M.

Now, for each $f \in \{1, \ldots, n\}$ denote as $\mathbf{b_f}$ the solution obtained by solving the system $M\mathbf{x} = \mathbf{0}$ where all variables corresponding to free columns are set to 0, except for $x_{F(f)}$ which is set to 1. Note that this uniquely determines the value of pivot variables since M is in RREF; thus

$$x_{P(p)} = \sum_{f=1}^{n} -M_{pF(f)} x_{F(f)}, \forall p \in \{1, \ldots, m-n\} \tag{3.6}$$

Lemma 3.8. *$\mathcal{B} = \{\mathbf{b_f} \mid f \in \{1, \ldots, n\}\}$ is a basis of ker(M). Moreover, $\forall \mathbf{v} \in$ ker(M) : $\mathbf{v} = \sum_{f=1}^{n} \mathbf{v}_{F(f)} \mathbf{b_f}$.*

Proof. **Linear independence:** For each $f \in \{1, \ldots, n\}$, $\mathbf{b_f}$ has 1 at position $F(f)$ while all other elements of $\mathcal{B}$ have 0 at position $F(f)$. Consequently, $\mathbf{b_f}$ cannot be expressed as a linear combination of other elements of $\mathcal{B}$.

Generators: We show that each vector $\mathbf{v} \in$ *ker*(M) can be written as a linear combination of elements of $\mathcal{B}$. More precisely, we show that $\mathbf{v} = \sum_{f=1}^{n} \mathbf{v}_{F(f)} \mathbf{b_f}$.

(a) For each $f \in \{1, \ldots, n\} : \mathbf{v}_{F(f)} = \sum_{\hat{f}=1}^{n} \mathbf{v}_{F(\hat{f})} \mathbf{b}_{\hat{\mathbf{f}}F(f)}$ as $\mathbf{b}_{\mathbf{f}F(f)} = 1$ and $\mathbf{b}_{\hat{\mathbf{f}}F(f)} = 0$ for $\hat{f} \neq f$.

(b) Fix a pivot index $p \in \{1, \ldots, m-n\}$. To see that $\mathbf{v}_{P(p)} = \sum_{f=1}^{n} \mathbf{v}_{F(f)} \mathbf{b}_{\mathbf{f}P(p)}$, note that $\mathbf{v}$ and all elements of $\mathcal{B}$ are solutions to the system $M\mathbf{x} = \mathbf{0}$, so they satisfy Equation (3.6). Instantiating Equation (3.6) with $\mathbf{b_f}$ for $f \in \{1, \ldots, n\}$ we get

$$\mathbf{b}_{\mathbf{f}P(p)} = \sum_{\hat{f}=1}^{n} -M_{pF(\hat{f})} \mathbf{b}_{\mathbf{f}F(\hat{f})} = -M_{pF(f)} \tag{3.7}$$

since $\mathbf{b}_{\mathbf{f}F(\hat{f})} = 1$ when $\hat{f} = f$ and 0 otherwise. Now, $\mathbf{v}_{P(p)} = \sum_{f=1}^{n} \mathbf{v}_{F(f)} \mathbf{b}_{\mathbf{f}P(p)}$ is obtained by instantiating Equation (3.6) with $\mathbf{v}$ and then replacing $-M_{pF(f)}$ by $\mathbf{b}_{\mathbf{f}P(p)}$ using Equation (3.7).

Combining (a) and (b), we have shown that $\mathbf{v}$ can be expressed as a linear combination of $\mathcal{B}$, which together with the linear independence of $\mathcal{B}$ concludes the proof. $\square$

A direct consequence of Lemma 3.8 is that the *coordinates* of $\mathbf{v} \in \mathit{ker}(M)$ with respect to basis $\mathcal{B}$, i.e., the coefficients of elements of $\mathcal{B}$ in the linear combination expressing $\mathbf{v}$, are positive if $\mathbf{v} > 0$. These coordinates are denoted as $\alpha_1, \ldots, \alpha_n$ in Algorithm 3.1, and we have just shown that using this standard algorithm for the computation of a kernel's basis the coordinates are guaranteed to be positive (line 6). However, the elements of the basis $\mathcal{B}$ are not guaranteed to be non-negative vectors.

Ensuring non-negativity of the basis. The second part of the algorithm, the loop on lines 7-12, modifies the elements of the basis. It gradually makes all elements non-negative, while at the same time it keeps the coordinates of vector $\mathbf{v}$, corresponding to the current basis, positive. Given an element of the basis $\mathbf{b_i}$ such that its j-th element is negative, the algorithm replaces the element $\mathbf{b_i}$ with a new element $\mathbf{b_i'} := \mathbf{b_i} + \frac{-\mathbf{b_i}_j}{\mathbf{v}_j}\mathbf{v}$. After replacing $\mathbf{b_i}$ with $\mathbf{b_i'}$, the resulting set of vectors is still a basis of $\mathit{ker}(M)$.

Lemma 3.9. *The set of vectors $\mathcal{B}' = (\mathcal{B} \setminus \{\mathbf{b_i}\}) \cup \{\mathbf{b_i'}\}$ is a basis of $\mathit{ker}(M)$.*

Proof. We show that $\mathbf{b_i}$ can be expressed as a linear combination of vectors from $\mathcal{B}'$. This is sufficient to show that $\mathcal{B}'$ consists of linearly independent vectors and that it generates $\mathit{ker}(M)$. Let us denote the constant $\frac{-\mathbf{b_i}_j}{\mathbf{v}_j}$ as K and note that $K > 0$ since $\mathbf{v}_j > 0$ and $\mathbf{b_i}_j < 0$. We first express $\mathbf{b_i'}$ as

$$\begin{aligned}\mathbf{b_i'} = \mathbf{b_i} + K\mathbf{v} = \mathbf{b_i} + K \sum_{f=1}^{n} \alpha_f \mathbf{b_f} \\ = \mathbf{b_i}(1 + K\alpha_i) + K \sum_{f \neq i} \alpha_f \mathbf{b_f}\end{aligned}$$

and now $\mathbf{b_i}$ can be expressed as a linear combination of elements of $\mathcal{B}'$:

$$\mathbf{b_i}(1+K\alpha_i) = \mathbf{b_i'} - K\sum_{f\neq i}\alpha_f\mathbf{b_f}$$
$$\mathbf{b_i} = \frac{\mathbf{b_i'} + \sum_{f\neq i} -K\alpha_f\mathbf{b_f}}{1+K\alpha_i}$$

□

After this replacement, (at least) one negative value has been successfully eliminated: As $K > 0$ and $\mathbf{v} > \mathbf{0}$, it follows that $\mathbf{b_i'} > \mathbf{b_i}$ and $\mathbf{b_i'}_j = 0$.

As the last step, we show that the new coordinates of $\mathbf{v}$ (with respect to the new basis) are still positive.

Lemma 3.10. *Let $\boldsymbol{\alpha}'$ denote the coordinates of $\mathbf{v}$ with respect to the new basis $\mathcal{B}'$. Then $\boldsymbol{\alpha}' > \mathbf{0}$.*

Proof. First, consider the result of a linear combination of the new basis $\mathcal{B}'$ with the old coefficients $\boldsymbol{\alpha}$:

$$\alpha_1\mathbf{b_1} + \ldots + \alpha_i\mathbf{b_i'} + \ldots + \alpha_n\mathbf{b_n} = \sum_{f=1}^{n}\alpha_f\mathbf{b_f} + \alpha_i K\mathbf{v} = \mathbf{v} + \alpha_i K\mathbf{v} = \mathbf{v}(1+\alpha_i K)$$

Now, set $C := 1 + \alpha_i K$ and note that $C > 1$ since $K > 0$ and $\alpha_i > 0$. It follows that

$$\mathbf{v} = \frac{\alpha_1}{C}\mathbf{b_1} + \ldots + \frac{\alpha_i}{C}\mathbf{b_i'} + \ldots + \frac{\alpha_n}{C}\mathbf{b_n}$$

and that $\boldsymbol{\alpha}' = \frac{\boldsymbol{\alpha}}{C}$ is the vector of coordinates of $\mathbf{v}$ with respect to the new basis $\mathcal{B}'$. Since $\boldsymbol{\alpha} > \mathbf{0}$ and $C > 0$, it follows that $\boldsymbol{\alpha}' > \mathbf{0}$ as required. □

We have shown that the loop on lines 7–12 preserves the invariant that the coordinates of $\mathbf{v}$ with respect to the current basis are all positive (lines 11,12) and that each iteration decreases the number of negative values of the basis vectors. As a result, Algorithm 3.1 terminates and returns a decomposition of the input vector $\mathbf{v}$ of size equal to the nullity of the input matrix M.

We first simulate the run of the algorithm on an example, then discuss its complexity and finally compare it to other approaches for computing interpolants as a conjunction of inequalities.

Example 3.11. *Consider an unsatisfiable system of inequalities $A \wedge B$ where $A = \{x_1+x_2 \leq 0, -x_1+x_3 \leq 0, x_1+x_4 \leq 0, -x_1+x_5 \leq 0\}$ and $B = \{-x_2-x_3-x_4-x_5 \leq -1\}$. The vector of Farkas coefficients witnessing the unsatisfiability of $A\wedge B$ is $\mathbf{k} = \begin{pmatrix}1 & 1 & 1 & 1 & 1\end{pmatrix}^\intercal$ and its restriction to A-part is $\mathbf{k_A} = \begin{pmatrix}1 & 1 & 1 & 1\end{pmatrix}^\intercal$. The only A-local variable is x_1, so the matrix of A-local coefficients is $L^\intercal = \begin{pmatrix}1 & -1 & 1 & -1\end{pmatrix}$. We simulate the run of Algorithm 3.1* *on $\mathbf{k_A}$ and $L^\intercal$: Since $L^\intercal$ is already in RREF, nothing changes on line 1. Now, the rank of $L^\intercal$ is 1 and it has 4 columns, thus its nullity is 3 and we can compute a decomposition of $\mathbf{k_A}$ of size 3. The first column of $L^\intercal$ is pivot while the other three columns are free. The computation of the initial basis of $ker(L^\intercal)$ (line 4) yields three vectors:*

$$b_1 = \begin{pmatrix}1\\1\\0\\0\end{pmatrix}, \quad b_2 = \begin{pmatrix}-1\\0\\1\\0\end{pmatrix}, \quad b_3 = \begin{pmatrix}1\\0\\0\\1\end{pmatrix}.$$

The coordinates of $\mathbf{k_A}$ with respect to this basis is $\boldsymbol{\alpha} = \begin{pmatrix}1 & 1 & 1\end{pmatrix}^\intercal$. As $\mathbf{b_2}_1 < 0$ we enter the loop on line 7 where the new vector $\mathbf{b_2'}$ is computed as $\mathbf{b_2'} = \mathbf{b_2} + \mathbf{k_A} = \begin{pmatrix}0 & 1 & 2 & 1\end{pmatrix}^\intercal$. Then, the coordinates are divided by a constant $C = 2$ to obtain the new coordinates $\boldsymbol{\alpha} = \begin{pmatrix}1/2 & 1/2 & 1/2\end{pmatrix}^\intercal$. Since there are no more negative elements in the vectors of the basis, the decomposition $\mathbf{k_A} = 1/2 * \begin{pmatrix}1 & 1 & 0 & 0\end{pmatrix} + 1/2 * \begin{pmatrix}0 & 1 & 2 & 1\end{pmatrix} + 1/2 * \begin{pmatrix}1 & 0 & 0 & 1\end{pmatrix}$ is returned. This decomposition results in the decomposed interpolant

$$I^{Dec} = (x_2 + x_3 \leq 0) \wedge (x_3 + 2x_4 + x_5 \leq 0) \wedge (x_2 + x_5 \leq 0).$$

Complexity of Algorithm 3.1. Considering the matrix of A-local coefficients L for m inequalities and l A-local variables, the algorithm runs on matrix $M = L^\intercal$ with m columns and l rows. When the transformation of M to RREF is done by Gauss-Jordan elimination, it needs to perform $\mathcal{O}(m^2 l)$ arithmetic operations. After the transformation, the number of (non-zero) rows is r, which is the rank of M and we know that $r \leq l$. With n denoting the nullity of M, Rank-Nullity Theorem implies that $r + n = m$ and consequently that $n < m$. The complexity of the computation of an initial basis is $\mathcal{O}(nm)$ since we are computing n basis vectors, each of size m. Determining the value for every element of each basis vector is immediate: it is 0 or 1 for positions corresponding to the free columns, and it is a negated coefficient from RREF(M) for positions corresponding to the pivot columns, see Equation (3.7). Finally, one iteration of the loop that ensures non-negativity of the basis needs just $\mathcal{O}(m)$ arithmetic operations and the termination can be ensured after $\mathcal{O}(n)$ iteration. To see this, note that a basis vector $\mathbf{b_i}$ can be made non-negative in one iteration when the index j is used that maximizes $\frac{-\mathbf{b_i}_j}{\mathbf{v}_j}$. The whole loop thus requires $\mathcal{O}(nm)$ arithmetic operations. The complexity of the algorithm is thus dominated by the first part—computing RREF of the input matrix.

3.4.3 Comparison with Other Approaches

Given an unsatisfiable system of inequalities (A, B), Cimatti et al. [61] recognized two extreme points in the spectrum of possible interpolants. On one side, there is the Farkas interpolant in the form of single inequality obtained as a weighted sum of inequalities from A with weights given by Farkas coefficients. On the other side, it is possible to employ

$$\begin{tikzcd}
1\times(x_1+x_2\leq 0) \arrow[d] & 1\times(-x_1+x_3\leq 0) \arrow[dl] \\
x_2+x_3\leq 0 \arrow[d] & 1\times(x_1+x_4\leq 0) \arrow[dl] \\
x_1+x_2+x_3+x_4\leq 0 \arrow[d] & 1\times(-x_1+x_5\leq 0) \arrow[dl] \\
x_2+x_3+x_4+x_5\leq 0 \arrow[d] & 1\times(-x_2-x_3-x_4-x_5\leq -1) \arrow[dl] \\
0\leq -1 &
\end{tikzcd}$$

Figure 3.2. Proof of unsatisfiability of the system from Example 3.11.

quantifier elimination to compute the strongest possible interpolant for (A, B) which will result in a conjunction of inequalities (if possible). If all A-local variables are existentially quantified in A and eliminated, then this is guaranteed to yield an interpolant. However, as Cimatti et al. note, quantifier elimination is potentially a very expensive operation.[3] Therefore, they propose modifications to the procedure computing the interpolant from the proof of unsatisfiability. The observation they make is that the only purpose of the summation of inequalities when traversing the proof is to eliminate A-local variables. If the leaves of the proof do not contain A-local variables, no summation is needed, and the conjunction of the inequalities in the leaves is already an interpolant. This corresponds to our notion of *trivial elements* of the decomposition. Based on this observation, they proposed a modification to the proof-based algorithm that performs *only the summations that are necessary for eliminating A-local variables.*

Example 3.12. *Consider the unsatisfiable system of inequalities from Example 3.11. Figure 3.2 shows a possible proof of unsatisfiability according to the description of [61].*

The computation of Farkas interpolant as described by Equation (3.1) *can be simulated by replacing the leaves from B with $0 \leq 0$. The resulting Farkas interpolant is*

$$I^F = x_2 + x_3 + x_4 + x_5 \leq 0.$$

Applying the modification from [61] avoids one unnecessary sum and results in an interpolant

$$I^M = (x_2 + x_3 \leq 0) \wedge (x_4 + x_5 \leq 0).^4$$

As seen in Example 3.11, our approach yields interpolant with three conjuncts

$$I^{Dec} = (x_2 + x_3 \leq 0) \wedge (x_3 + 2x_4 + x_5 \leq 0) \wedge (x_2 + x_5 \leq 0).$$

[3]Even when restricted to conjunction of inequalities, as is our case. For example, in Fourier-Motzkin procedure eliminating one variable can increase the number of inequalities from m to $m^2/4$ in the worst case. Thus, eliminating n variables increases the number of inequalities to $4(\frac{m}{4})^{2^n}$ in the worst case.

[4]This is indeed the interpolant computed by MathSAT 5.6.0

Finally, existentially quantifying x_1 in A and eliminating this quantifier yields interpolant with four conjuncts

$$I^{QE} = (x_2 + x_3 \leq 0) \wedge (x_2 + x_5 \leq 0) \wedge (x_3 + x_4 \leq 0) \wedge (x_4 + x_5 \leq 0).$$

Note that I^{QE} is the strongest and I^F is the weakest interpolant in this quadruple, while I^M and I^{Dec} are incomparable in terms of logical strength. However, the advantage of our algorithm is that even though its result depends on the order of the inequalities (the order of columns of $L^\intercal$), it guarantees to find a decomposition of size 3 in our example. If the first and third inequalities are switched, the decomposed interpolant computed by Algorithm 3.1 is

$$I^{Dec'} = (x_4 + x_3 \leq 0) \wedge (x_3 + 2x_2 + x_5 \leq 0) \wedge (x_4 + x_5 \leq 0)$$

while if the first and second inequalities are switched, the computed interpolant is

$$I^{Dec''} = (x_2 + x_4 + 2x_5 \leq 0) \wedge (x_3 + x_4 \leq 0) \wedge (x_3 + x_2 \leq 0).$$

On the other hand, the approach of [61] is, in some sense, even more sensitive to the order of the input inequalities (the shape of the proof) since the order can influence the size of the decomposition. If the second and the third inequalities are switched, then their approach does not detect the opportunity for decomposition and returns the Farkas interpolant I^F. Our algorithm in this situation returns an interpolant equivalent to I^{Dec}.

3.5 Experiments

We have implemented the computation of decomposed interpolants and their duals using Algorithm 3.1 in our SMT solver OpenSMT [122], which already provided a variety of interpolation algorithms for propositional logic [124, 171], theory of uninterpreted functions [7] and theory of linear real arithmetic [8].

We evaluated the effect of decomposed interpolants in a model-checking scenario using the model checker sally [129] with Yices [83] for satisfiability queries and OpenSMT for interpolation queries[5]. We experimented with four LRA interpolation algorithms: the original interpolation algorithms based on Farkas' lemma, (i) Itp_F and (ii) Itp'_F, and the interpolation algorithm computing decomposed interpolants, (iii) Itp_{DI} and (iv) Itp'_{DI}. OpenSMT computes interpolants from the proof of unsatisfiability. In this approach, the interpolants computed for LRA conflicts are combined based on interpolation rules for propositional logic and the structure of the proof. In our experiments, we fixed the propositional part of the interpolation algorithm to use McMillan's interpolation rules [155]. We split our analysis of the experiments into two parts. In Section 3.5.1, we

[5]Detailed description of the set-up and specifications of the experiments, together with all the results, can be found at `http://verify.inf.usi.ch/content/decomposed-interpolants`

	Itp_F		Itp'_F		Itp_{DI}		Itp'_{DI}	
Problem set	solved (V/I)	time(s)	solved (V/I)	time(s)	solved (V/I)	time(s)	solved (V/I)	time(s)
approxagree (9)	9 (8/1)	127	9 (8/1)	138	**9 (8/1)**	**106**	9 (8/1)	126
azadmanesh (20)	**20 (17/3)**	**418**	20 (17/3)	639	20 (17/3)	422	20 (17/3)	1,202
cav12 (99)	**68 (48/20)**	**2,097**	67 (48/19)	2,580	66 (48/18)	1,441	66 (47/19)	2,446
conc (6)	3 (3/0)	20	3 (3/0)	22	**5 (5/0)**	**313**	3 (3/0)	21
ctigar (110)	**74 (54/20)**	**3,066**	70 (50/20)	1,919	71 (51/20)	3,077	58 (39/19)	1,701
hacms (5)	2 (2/0)	332	**2 (1/1)**	**251**	1 (1/0)	5	1 (1/0)	5
lfht (27)	17 (17/0)	319	18 (18/0)	448	**22 (22/0)**	**2,784**	16 (16/0)	26
lustre (790)	**773 (437/336)**	**3,530**	769 (436/333)	3,180	766 (433/333)	3,990	741 (416/325)	2,021
misc (10)	8 (7/1)	154	8 (7/1)	127	**9 (7/2)**	**57**	9 (7/2)	888
om (9)	9 (7/2)	6	**9 (7/2)**	**4**	9 (7/2)	6	**9 (7/2)**	**4**
ttastartup (3)	**2 (1/1)**	**325**	1 (1/0)	7	1 (1/0)	11	1 (1/0)	15
ttesynchro (6)	**6 (3/3)**	**10**	6 (3/3)	11	6 (3/3)	13	6 (3/3)	13
unifapprox (11)	11 (8/3)	71	**11 (8/3)**	**64**	11 (8/3)	71	11 (8/3)	448
Total (1,105)	**1,002 (612/390)**	**10,475**	993 (607/386)	9,390	996 (611/385)	12,296	950 (573/377)	8,916

Table 3.1. Performance of SALLY using different interpolation algorithms of OPENSMT

analyse the performance of the model checker using different LRA interpolation algorithms. We focus specifically on a detailed comparison of Itp_F and Itp_{DI}, i.e., the default algorithm and our proposed algorithm. In Section 3.5.2, we analyse the performance of a portfolio of interpolation algorithms and measure the contribution of our proposed algorithm. For comparison, we also run a version of SALLY using MATHSAT as the interpolation engine and compare to the contribution of the decomposing algorithm proposed in [61].

The experiments were run on a large set of benchmarks consisting of several problem sets related to fault-tolerant algorithms (**azadmanesh**, **approxagree**, **om**, **hacms**, **misc**, **ttesynchro**, **ttastartup**,**unifapprox**), software model checking (**cav12**, **ctigar**), simple concurrent programs (**conc**), and a lock-free hash table (**lfht**). A benchmark suite of the KIND model checker is also included (**lustre**). Each benchmark is a transition system with formulas characterizing initial states, a transition relation and a property that should hold. SALLY can finish with two possible answers (or run out of resources with no answer): *valid* means the property holds and an invariant implying the property has been found; *invalid* means the property does not hold and a counterexample leading to a state where the property does not hold has been found. In the plots, we denote the answers as $+$ and $\circ$, respectively. The benchmarks were run on Linux machines with the Intel E5-2650 v3 processor (2.3 GHz) and 64GB of memory. Each benchmark was restricted to 600 seconds of running time and to 4GB of memory.

3.5.1 Comparing Individual Configurations

Table 3.1 presents the results of the model checker's runs using different interpolation algorithms. The results are summarized *by category* with the name of the category and the number of corresponding benchmarks in the first column. The two columns per interpolation algorithm show the number of benchmarks solved successfully (validated/invalidated) within the resource limits and the total running time for the *solved* benchmarks.

The results suggest that Itp_F interpolation algorithm achieves the best result overall. However, there are certain cases where Itp_{DI} is faring better, for example the **lfht** category. Before we present a more thorough comparison between these two algorithms we note that the configuration using Itp'_{DI}, which computes the weakest interpolants, performs very poorly compared to the others. Closer inspection revealed that it did not solve any benchmarks not solvable by other configurations. It did solve a few benchmarks faster than others, but the improvement was negligible. On the other hand, the overall drop in performance is large. We conclude that computing very weak interpolants is a bad strategy in this model-checking scenario.

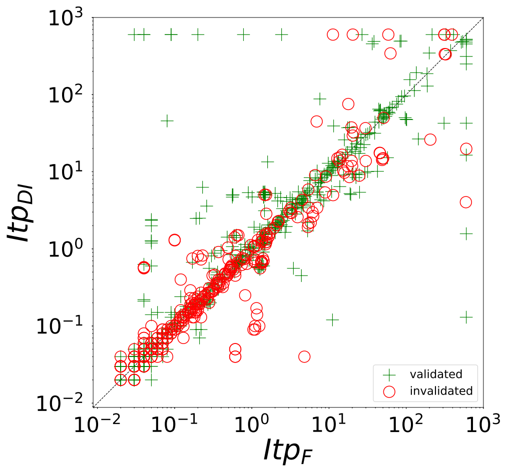


Figure 3.3. Evaluation of the decomposed interpolants in model checking scenario: comparison of performance of SALLY using OPENSMT with different interpolation procedures, Itp_F and Itp_{DI}.

As mentioned before, the results summarized in Table 3.1 suggest that Itp_F performs better than Itp_{DI} overall. However, a closer look reveals that the situation is more complicated. Figure 3.3 illustrates a direct comparison between these two algorithms. Each point represents one benchmark, x-axis corresponds to the runtime (in seconds) of SALLY using Itp_F as the interpolation algorithm in OPENSMT, and y-axis corresponds to the runtime of SALLY using Itp_{DI}. The direct comparison shows that in some cases the use of decomposed interpolants outperforms the original procedure, sometimes by

	Itp_F		Itp_{DI}	
benchmark	solved	avg. time	solved	avg. time
fib__benc__safe__v1	0	-	100	46.5
fib__benc__safe__v2	0	-	100	0.01
dillig01.c	0	-	100	0.1
dillig03.c	0	-	100	0.1
lifnat.c	17	510	29	471
lfht__2__mini__cleaned.prop1	21	362	57	344
lfht__2__mini__lemma5c	18	257	69	293
lfht__2__mini__lemma5e	0	-	30	347
lfht__2__mini__lemma5f	1	188	39	363
lfht__2__mini__lemma5g	22	284	47	311
DRAGON__12__e2__1618__e2__138	99	25	100	19
mvs__with__timeouts3	73	251	98	64

Table 3.2. Aggregated results from 100 runs of the model checker on selected benchmarks

an order of magnitude. Even though Itp_{DI} solved 6 benchmarks less than Itp_F, it still managed to solve 12 benchmarks that Itp_F was not able to solve within the resource limits. Moreover, on a common set of non-trivial (runtime at least 10 seconds) solved benchmarks, it improved the performance by more than 10% on 45 benchmarks (out of 116 such benchmarks).

During the evaluation, we realized that a small modification in the SMT solver sometimes had a huge effect on the performance of the model checker. It made previously unsolved instance easily solvable or the other way around. To confirm that using Itp_{DI} is indeed better than using Itp_F for particular benchmarks, we ran an additional set of experiments. For each of the 12 benchmarks solved by Itp_{DI} but not solved by Itp_F we ran the model checker 100 times, each time with a different random seed for the interpolating solver. The results are summarized in Table 3.2. For each of the two configurations, the table reports how many runs (out of 100) of the model checker finished successfully within the resource limits and the average time of the successful runs. This experiment demonstrates that there are indeed benchmarks where the decomposition is necessary, while using the original Farkas algorithm leads to divergence. In other cases, the use of decomposed interpolants leads to a higher chance of a successful result and/or better runtime of the model checker. Note that these benchmarks were picked deliberately to confirm that Itp_{DI} performs better on them than Itp_F, based on our experiments on the whole benchmark set.

For the final aspect of the direct comparison of Itp_F and Itp_{DI} we collected statistics from the runs of SALLY with Itp_{DI} about how often Itp_{DI} manages to decompose the vector of Farkas coefficients, thus returning a different interpolant than Itp_F would. These results are summarized in Table 3.3. The second column reports the *number* of

	Itp_{DI}		
Problem set	#problems with some decomposition	#non-trivial itp problems	#decomposed itps
approxagree (9)	1 (1/1)	7	7 (4/3)
azadmanesh (20)	0 (0/0)	1,818	0 (0/0)
cav12 (99)	40 (30/29)	707,414	6,464 (747/5,719)
conc (6)	3 (3/3)	39,135	25,603 (4,030/21,033)
ctigar (110)	70 (58/69)	4,064,827	1,106,642 (61,371/1,049,904)
hacms (5)	5 (5/5)	424,532	32,331 (3,628/28,703)
lfht (27)	14 (14/14)	786,837	126,568 (5,464/121,104)
lustre (790)	327 (96/299)	2,916,829	2,001,503 (9,115/2,001,058)
misc (10)	8 (7/8)	59,266	12,054 (2,363/10,024)
om (9)	6 (6/0)	974	380 (380/0)
ttastartup (3)	3 (2/3)	117,303	12,165 (240/11,925)
ttesynchro (6)	4 (4/4)	90	90 (90/69)
unifapprox (11)	1 (1/0)	1	1 (1/0)

Table 3.3. Interpolation statistics. The numbers in parentheses count only situations where decomposition contains some trivial and some non-trivial elements (trivial/non-trivial).

benchmarks with at least a *single* decomposition (any; with at least one trivial element; with at least one non-trivial element). The third column reports the total number of interpolation problems for theory conflict, excluding those without even theoretical possibility for decomposition. There is no possibility for decomposition if all inequalities are from one part of the problem (resulting in trivial interpolants, either $\top$ or $\bot$) or there is only a single inequality in the A-part (trivially yielding an interpolant equal to that inequality). The last column reports the number of successfully decomposed interpolants (with at least one trivial element; with at least one non-trivial element). Note that it can happen that a successful decomposition contains both trivial and non-trivial elements. We see that at least one decomposition was possible in only less than half of all the benchmarks. This explains why there are many points on the diagonal in Figure 3.3. On the other hand, it shows that the test for the *possibility* of decomposition is cheap and does not represent a significant overhead. Another conclusion we can draw is that when the structure of the benchmark enables decomposition, it can often be discovered in many theory conflicts that appear during the solving.

3.5.2 Analysis of the Portfolio

In this part, we present yet another way to measure the contribution of the decomposed interpolants: the contribution to the virtual best configuration. We consider a virtual portfolio consisting of configurations of SALLY using different interpolation algorithms of OpenSMT. In addition, we also consider a separate virtual portfolio of configurations

	config.	#uniq. solved	PAR-2 regret	
OpenSMT	Itp_F	4	4046	3.5%
	Itp'_F	3	4586	3.9%
	Itp_{DI}	10	10245	8.8%
MathSAT	Itp_F	0	260	0.2%
	Itp'_F	3	3594	3.3%
	Itp_M	6	7754	7%

Table 3.4. Contribution of the configurations to their respective portfolios.

of SALLY using MathSAT. The result of a virtual portfolio on a benchmark is the best result achieved by any of the configurations of the portfolio. As noted before, the configuration using Itp'_{DI} performed quite poorly on our benchmarks. Since MathSAT can compute Farkas interpolants and its duals, and restricted form of decomposed interpolants but not its dual, we also exclude Itp'_{DI} from the portfolio of OpenSMT's configurations, with minimal impact on the performance. We denote the heuristic for computing decompositions described in [61] and available in MathSAT as Itp_M. We use the number of solved instances and PAR-2 score as a metric of measuring the performance. PAR-2 score is computed as the sum of runtime on solved instances plus two times the timeout for each unsolved instance. Finally, for each configuration we compute the number of uniquely solved instances (not solved by any other configuration in the portfolio) and *regret*, i.e., how much would the PAR-2 score of the portfolio worsen, if that particular configuration was excluded from the portfolio. The results are summarized in Table 3.4. Note that OpenSMT and MathSAT portfolios are considered separately.

OpenSMT configuration portfolio is able to solve 1,017 benchmarks with PAR-2 score 116,117. MathSAT configuration portfolio is able to solve 1,018 benchmarks with PAR-2 score 110,356. We hypothesize that the better performance of MathSAT can be at least partially attributed to the fact that it supports interpolation in combination with incremental solving while OpenSMT does not. In both portfolios, the ability to compute decomposed interpolants (even in the restricted form) significantly improves the performance of the portfolio. We also see that the contribution of our algorithm based on methods from linear algebra to OpenSMT portfolio is slightly larger than the contribution of the heuristic Itp_M to the MathSAT portfolio. Additionally, our algorithm solves more instances uniquely within its portfolio. Interestingly, the contribution of the configuration computing Farkas interpolants is non-trivial in OpenSMT, but almost non-existent in MathSAT. Our hypothesis is that Itp_M, compared to Itp_{DI}, decomposes less often and the decompositions are of smaller size (e.g., in the situation from Example 3.12). This would mean that the interpolants from Itp_M are more often similar (or even identical) to Farkas interpolants, which would make the MathSAT portfolio less diverse than the OpenSMT portfolio.

3.6 Related Work

The possible weakness of Farkas interpolants for use in model checking was recognized in [179]. The authors demonstrate that Farkas interpolation does not satisfy the condition needed for proving convergence of a model-checking algorithm pd-kind [129]. Indeed, the model checker sally [129], which implements pd-kind, diverges on our example from Section 3.2 if Farkas interpolation is used in its underlying interpolation engine. To resolve this problem [179] introduces a new interpolation procedure that guarantees the convergence of a special sequence of interpolation problems often occurring in model checking problems. However, this interpolation algorithm is based on a decision procedure called conflict resolution [138], which is not as efficient as the Simplex-based decision procedure used by most state-of-the-art SMT solvers. In contrast, we show how the original Simplex-based decision procedure using Farkas coefficients can be modified to produce interpolants not restricted to the single-inequality form, while additionally obtaining strength guarantees with respect to the original Farkas interpolants.

The reasoning engine Spacer [137] was also known to be affected by this weakness of Farkas interpolants. The verification framework SeaHorn [107], which relies on Spacer, originally used to obtain additional invariants from abstract interpretation to avoid the divergence. Recently, the algorithm in Spacer was enriched with global guidance [190]. One part of the global guidance is monitoring the progress of the model checker, detecting an emergent diverging behaviour and applying a special rule to prevent the divergence.

A different approach to control the interpolants at the level of the model checking algorithm is *interpolation abstraction* [177]. It is a powerful technique for restricting interpolants to conform to a prescribed form. It enables fine-grained control over symbol occurrences in an interpolant. The abstraction is expected to be provided by the application, and a reasonable choice has been given for software model checking. The disadvantage is that it requires auxiliary variables to be added to the original interpolation problem to enforce the abstraction on the interpolant computation.

Besides the application in interpolation-based model checking, the interpolation itself has received significant attention in the last two decades. The work on LRA interpolation dates back to 1997 [166]. A compact set of rules for deriving LRA interpolants from the proof of unsatisfiability in an inference system was presented in [156]. The interpolants in these works were the Farkas interpolants. Current methods usually compute Farkas interpolants from explanations of unsatisfiability extracted directly from the Simplex-based decision procedure inside the SMT solver [84]. Recent investigations of controlling the strength of LRA interpolants showed that the information from primal and dual Farkas interpolants could be combined to obtain an interpolant of intermediate strength [8].

There is an infinite family of interpolants between a primal and a dual interpolant, and the strength can be controlled with a single *strength factor*. However, these interpolants are still restricted to single inequalities and are always weaker than the primal Farkas interpolant.

The first discussion on how to obtain interpolants in the form of a conjunction of inequalities from Farkas coefficients is present in [61]. However, their approach is based on a simple heuristic which does not discover the possibility for decompositions in some cases where our approach finds the decomposition easily. Moreover, their focus was purely on the interpolation techniques, and they did not discuss the application in model checking. We provided a detailed comparison to our approach in Section 3.4.3.

Besides the computation of interpolants from refutation proof, linear programming (LP) methods have been successfully used to compute interpolants [178]. LP solvers were used to compute *simple*, or even *beautiful* LRA interpolants [3, 180]. In [3], the authors use linear programming (LP) solver to check for the existence of a common half-plane interpolant for increasingly larger subparts of the given LRA problem. In [180], the authors use a similar method but only after the refutation proof has been constructed by standard solving methods. Our focus is not on the overall interpolant but on a single LRA conflict. However, in the context of interpolants from proofs produced by SMT solvers, our approach also has the potential for re-using components of interpolants for LRA conflicts across the whole proof.

Orthogonal to the studies of interpolation algorithms for LRA conflicts is the large body of work on the propositional part of interpolation procedures, e.g., [6, 81, 108, 125, 172, 197].

3.7 Conclusion and Future work

In this chapter, we have contributed a new interpolation algorithm for conflicts in the theory of linear real arithmetic. This algorithm generalizes the interpolation algorithm based on Farkas' lemma used in modern SMT solvers; it uses methods from linear algebra to identify linearly indepedent components and decompose Farkas interpolant. We showed that the algorithm is able to compute interpolants in the form of a *conjunction* of inequalities that are logically stronger than the single inequality returned by the original approach. This becomes useful in the IC3-style model-checking algorithms where Farkas interpolants have been shown to be a source of incompleteness. In our experiments, we have demonstrated that the opportunity to decompose Farkas interpolants frequently occurs in practice and that the decomposition often leads to (i) lower solving time and, in some cases, to (ii) solving a problem not solvable by the previous approach.

An interesting future research would be to go beyond a simple portfolio approach and automatically determine what kind of interpolant would be more useful for the current interpolation query in (not only) IC3-style model-checking algorithms.

Chapter 4

Transition Power Abstraction

Automated formal verification by means of model checking is popular because of the ability to both (1) find error paths for unsafe systems, and (2) *prove* the absence of error paths for safe systems. Recent techniques based on Satisfiability Modulo Theories (SMT), as well as the continuing improvements of SMT solvers [11, 60, 72, 83, 122], enable scalable applications of model checking to software verification [22]. Specifically, the idea of building a safe inductive invariant incrementally—pioneered by the hardware model checking algorithm IC3/PDR [41, 85]—has been successfully applied in several IC3-inspired approaches [58, 59, 93, 113, 129, 137], thus improving the capabilities of verification tools significantly.

Although this progress is undeniably encouraging, model checking still suffers from scalability issues associated with an exhaustive exploration of a system's states. For many systems, a large set of states must be observed to eventually detect a counterexample or synthesize an invariant.

The basic template for reachability-based analysis originated with *bounded model checking* (BMC) [29]. A typical BMC algorithm searches for counterexamples reachable in a finite number of steps, and if nothing is found, it increases the search limits and restarts. Most modern model-checking algorithms based on reachability analysis have adopted this philosophy because one of the advantages of this approach is that it finds the shortest counterexample (if one exists). However, it also results in scalability issues. Specifically, in modern software systems, it is not uncommon that a program must iterate through a particular loop thousands of times (or more) before it reaches some error state. These *deep* counterexamples pose problems for reachability-based algorithms that rely on unrolling the bounds of the system's transition relation one transition at a time.

In this chapter, we present the concept of *Transition Power Abstraction* (TPA) sequence and a novel model-checking algorithm for safety properties of transition systems. One key feature of this algorithm is a shift from the focus on *states* and *state abstractions* to the focus on *transitions* and *transition abstractions* [164]. TPA sequence is a sequence of abstract relations that gradually summarize (in an over-approximating manner) an increasing number of steps of the transition relation. The distinguishing feature is that the summarized number of steps increases exponentially, not linearly. The algorithm uses the TPA sequence for answering bounded reachability queries about the system, but it also extends and refines this sequence based on the information learned. Using the TPA sequence, the algorithm can quickly focus on the essential part of the search space and not waste time examining short paths that cannot lead to a counterexample. At the same time, it can discover *transition* invariants of the system sufficient to prove the system's safety. In this chapter, we present the theoretical ideas and pseudocode of the algorithm. The implementation details and experiments are given in Chapter 5 as part of the presentation of our Horn solver GOLEM. In the experiments, we demonstrate that TPA can detect counterexamples beyond the capabilities of state-of-the-art model checkers due to their depth and that proving safety using transition invariants complements state-of-the-art techniques for proving safety by discovering safe inductive invariant.

4.1 Preliminaries

In this chapter, we study the problem of whether or not a safety property holds in a given transition system. The basic notions related to this problem were explained in Section 2.5. A key concept in this problem is *reachability*, i.e., the existence of a trace between states of the system. The transitions and reachability are captured symbolically by *binary relations* over the set of states. *Concatenation* of relations is used to define relations that represent reachability in a fixed number of steps. Given two relations $R_1(x, y)$ and $R_2(y, z)$, their concatenation $R = R_1 \circ R_2$ is a relation over x, z such that $R(x, z) \iff \exists y : R_1(x, y)$ and $R_2(y, z)$. In transition systems, we can define relations representing multiple steps of a transition relation. For example, $Tr^2(X, X'') \equiv Tr(X, X') \circ Tr(X', X'')$ relates pairs of states (s, t) such that t is reachable from s in *exactly* two steps of the transition relation Tr. We also write that $(s, t) \in Tr^2$. Existence of a counterexample (a trace from some initial to some bad state) of a fixed length l can be encoded as a satisfiability check of the formula

$$Init(X^{(0)}) \wedge Tr(X^{(0)}, X^{(1)}) \wedge Tr(X^{(1)}, X^{(2)}) \wedge \ldots \wedge Tr(X^{(l-1)}, X^{(l)}) \wedge Bad(X^{(l)}),$$

where $X^{(i)}$ is a state variable shifted i steps, "with i primes". A satisfying assignment determines $l + 1$ states such that the first one is an initial state, the last one is a bad state, and each successor can be reached from its predecessor by one step of the transition relation Tr. If there is no satisfying assignment, no trace of l steps from $Init$ to Bad exists.

4.1.1 State and Transition Inductive Invariants

A set of states S is an inductive invariant iff

- $Init(X) \wedge Tr(X, X') \implies S(X')$,

- $S(X) \wedge \mathit{Tr}(X, X') \implies S(X')$.

An inductive invariant is *safe* if it excludes all bad states. Proving safety of a transition system by discovering a safe inductive invariant is one of the most popular approaches, especially for infinite-state systems.

The idea of a safe inductive invariant can be lifted from states to transitions. Let Tr^* denote the reflexive transitive closure of Tr. Then Tr^* represents reachability in any number of steps (including 0) in the system. We say that a transition formula $T(X, X')$ is a *transition invariant* iff $\mathit{Tr}^* \subseteq T$, i.e., $\forall X, X'\ \mathit{Tr}^*(X, X') \implies T(X, X')$.[1]

A transition formula $T(X, X')$ is an *inductive* transition invariant iff

1. $\mathit{Id}(X, X') \subseteq T(X, X')$, and
2. either $T(X, X') \wedge \mathit{Tr}(X', X'') \implies T(X, X'')$
   or $\mathit{Tr}(X, X') \wedge T(X', X'') \implies T(X, X'')$.

Note that an inductive transition invariant T is indeed a transition invariant, i.e., it over-approximates Tr^*. This can be easily proved by induction. Suppose that $(s, t) \in \mathit{Tr}^*$, i.e., s can reach t. Consider the shortest trace from s to t. If its length is 0, then by condition 1, $(s, t) \in T$. Suppose that the length of the shortest trace is $l > 0$. Then there is t's predecessor p on the trace, such that p can be reached from s in $l - 1$ steps and $(p, t) \in \mathit{Tr}$. By the induction hypothesis, $(s, p) \in T$, and based on the first option in condition 2, we get that $(s, t) \in T$, which concludes the proof. For the second option in condition 2, it suffices to consider in the inductive step the first successor of s instead of the first predecessor of t.

Note, however, that the two versions of condition 2 are not equivalent, as witnessed by the following example.

Example 4.1. *Consider the following transition relation Tr and a transition formula T:*

$$\mathit{Tr}(x, y, x', y') \equiv y \geq 0 \wedge x' = x \wedge y' = y + x \tag{4.1}$$

$$T(x, y, x', y') \equiv y' \geq y \vee y' \geq x \tag{4.2}$$

It is not hard to verify that $\mathit{Tr} \circ T \subseteq T$, but $T \circ \mathit{Tr} \not\subseteq T$. For the first case, we need to show that $y \geq 0 \wedge x' = x \wedge y' = y + x \wedge (y'' \geq y' \vee y'' \geq x')$ implies $y'' \geq y \vee y'' \geq x$. We can do a case analysis for the disjunction in the antecedent: If $y'' \geq x'$ then $y'' \geq x$ since $x' = x$. If $y'' \geq y'$ then $y'' \geq x + y$ (since $y' = x + y$) and thus $y'' \geq x$ as $y \geq 0$.

For the second case, consider the following three states $(x = 0, y = 0)$, $(x' = -1, y' = 0)$ and $(x'' = -1, y'' = -1)$. Then $T(x, y, x', y')$ and $\mathit{Tr}(x', y', x'', y'')$ hold, so $(T \circ \mathit{Tr})(x, y, x'', y'')$ holds; but $T(x, y, x'', y'')$ does not hold.

[1] Note that our definition is slightly simpler than that of [164], as it only depends on the transition relation and not on the initial states of the system.

Similarly to inductive invariants, inductive transition invariants can be used as a proof rule. If T is an inductive transition invariant and T does not relate any initial state with a bad state, then T is *safe*. *If a safe inductive transition invariant exists, then the system is safe.* Checking whether a transition formula T can be formulated as a satisfiability query: T is safe iff $\mathit{Init}(X) \wedge T(X, X') \wedge \mathit{Bad}(X')$ is unsatisfiable.

This proof rule can be further strengthened by weakening the assumption. We consider the following notions:

Definition 4.2 (left- and right-grounded transition invariant)**.** *Let T be a transition formula. If $\mathit{Init} \lhd \mathit{Tr}^* \subseteq \mathit{Init} \lhd T$ then we say T is a left-grounded transition invariant. If $\mathit{Tr}^* \rhd \mathit{Bad} \subseteq T \rhd \mathit{Bad}$ then we say T is a right-grounded transition invariant.*

We say that a transition formula is a grounded transition invariant if it is either left-grounded or right-grounded. Note that grounded transition invariants are closely related to the state invariants in the traditional sense: If $T(X, X')$ is a left-grounded transition invariant then $\exists X : \mathit{Init}(X) \wedge T(X, X')$ is a state invariant, i.e., it over-approximates all states reachable from *Init*. Additionally, we obtain a symmetric concept: If $T(X, X')$ is a right-grounded transition invariant then $\exists X' : T(X, X') \wedge \mathit{Bad}(X')$ over-approximates all states backward reachable from *Bad*.

The concept of a safe inductive transition invariant can be extended to grounded transition invariants.

Definition 4.3. *Let T be a transition formula with $\mathit{Id} \subseteq T$. If $\mathit{Init} \lhd T \circ \mathit{Tr} \subseteq \mathit{Init} \lhd T$ then T is an* inductive left-grounded transition invariant. *If $\mathit{Tr} \circ T \rhd \mathit{Bad} \subseteq T \rhd \mathit{Bad}$ then T is an* inductive right-grounded transition invariant.

Observation 4.4. *If a transition formula T is an inductive grounded transition invariant, then it is indeed a grounded transition invariant according to Definition 4.2.*

Proof. The proof is analogous to the proof that an inductive transition invariant is indeed a transition invariant. We show only the case of the right-grounded invariant, the left-grounded case is analogous. We want to show that if $(s, t) \in \mathit{Tr}^*$ and $t \in \mathit{Bad}$ then $(s, t) \in T$. The proof proceeds by induction on the length l of the shortest path from s to t. The base case $l = 0$ holds because $\mathit{Id} \subseteq T$. For the inductive step, consider m, the successor of s on the path to t. By induction, it holds that $(m, t) \in T$. Since $(s, m) \in \mathit{Tr}$, it follows by the assumption on T that $(s, t) \in T$. □

Inductive grounded transition invariants can be used as a proof rule for safety in the same way as inductive transition invariants: If a safe inductive grounded transition invariant exists, then the system is safe.

4.2 Intuition behind Transition Power Abstraction

We explain the problem of slow progress in state-of-the-art algorithms and the intuition that led to TPA in a simple example of a multi-phase loop (generalized from [185] where

```
x = 0; y = N;
while(x < 2N){
  x = x + 1;
  if(x > N)
    y = y + 1;
}
assert(y != 2N);
```

$$\begin{aligned} \mathit{Init}(x,y) &\equiv x = 0 \land y = N \\ \mathit{Tr}(x,y,x',y') &\equiv x < 2N \land x' = x + 1 \\ &\quad \land y' = \mathit{ite}(x' > N, y+1, y) \\ \mathit{Bad}(x,y) &\equiv x \geq 2N \land y = 2N \end{aligned}$$

Figure 4.1. An example of unsafe multi-phase loop

`N=50`), given in Figure 4.1. The program source code is given on the left, while the corresponding transition system is given on the right. Note that `N` represents a parameter of the system, not a nondeterministic variable.

Since the assertion is placed after the loop, any counterexample requires finding a complete unrolling of the loop, i.e., all `2N` iterations (or `2N` steps in the corresponding transition system). Interestingly, even a linear growth of `N` results in the exponential growth of complexity of the search for counterexamples. Because of the control-flow divergence in each iteration of the loop, the number of possible program paths (that a verifier explores) doubles with each increment of counter `x`. Consider a run of BMC on this example: Using SMT queries, it tests the existence of a counterexample of length $0, 1, 2, \ldots, 2N$. All queries except the last one are unsatisfiable since the loop requires $2N$ iterations to finish and reach the assertion. Thus BMC requires $2N$ SMT queries; however, the queries get more and more complex. Each new query contains an additional copy of the state variables, compared to the previous query.

This inefficient exploration of a state space caused by the slow increment of the considered bound was the primary motivation for our search for an alternative approach. We were inspired by the technique known as *binary exponentiation* or *exponentiation by squaring* used in many areas of computer science (see, e.g., [161]). We can apply this idea to a transition relation in the following way: Let $R^{\leq n}$ denote a relation of reachability in $\leq 2^n$ steps. The sequence of relations $R^{\leq}$ can be defined inductively as

$$\begin{aligned} R^{\leq 0} &= \mathit{Id} \cup \mathit{Tr}, \\ R^{\leq n+1} &= R^{\leq n} \circ R^{\leq n}. \end{aligned} \tag{4.3}$$

Note, however, that when representing these relations as formulas, the inductive step would require quantification over the intermediate states, to keep the relation expressed only over the source and target state variables, i.e., $R^{\leq n+1}(X, X') \equiv \exists Y\ R^{\leq n}(X, Y) \land R^{\leq n}(Y, X')$. Thus, quantifier elimination is required to avoid the accumulation of many copies of state variables. One option is to apply quantifier elimination eagerly. A much more common approach in verification is to use *Craig interpolation* to *over-approximate* quantifier elimination and only lazily refine the abstraction on demand. The *transition power*

```
input  : transition system $\mathcal{S} = \langle Init, Tr, Bad \rangle$
global : TPA sequence $ATr^{\leq 0}, \ldots, ATr^{\leq n}, \ldots$ (lazily initialized to $true$)
Function CheckSafetyTPA($\langle Init, Tr, Bad \rangle$):
1   $ATr^{\leq 0} \leftarrow Id \vee Tr$;   $n \leftarrow 0$
2   while $TRUE$ do
3       if IsReachable($n, Init, Bad$) $\neq \emptyset$ then return UNSAFE
4       if HasInvariant($\mathcal{S}, n$) then return SAFE
```

Algorithm 4.1. Main procedure for checking safety

abstraction sequence captures the idea outlined above. Intuitively, the elements of the sequence are formulas that over-approximate reachability in an exponentially increasing number of steps of the transition relation. Moreover, they are always expressed only over two copies of the state variables.[2] For our example in Figure 4.1, first $\log_2 N$ elements could be $x' \leq x + 1, x' \leq x + 2, x' \leq x + 4, x' \leq x + 8, \ldots, x' \leq x + 2^{\log_2 N}$. We show how to build a procedure for checking safety properties of transition systems based on the idea of the TPA sequence.

4.3 Transition Power Abstraction for Checking Safety Properties

First, we present a simpler version of the algorithm for checking safety properties of transition systems using the TPA sequence; later, we give an improved but more complex version. The simpler version allows us to better explain the fundamental concepts of the algorithm: answering bounded reachability queries with the TPA sequence, refinement of the TPA sequence, and checks for inductive invariants.

Our main procedure—given in Algorithm 4.1—follows the typical scheme of bounded model checking where, in each iteration, the reachability of *Bad* is checked within a certain *bounded* number of steps, and the bound gradually increases. This scheme has also been adopted by other model-checking algorithms, such as Spacer [137] and interpolation-based model checking [91, 155, 194], which further support a generalization/adaptation of the proof of bounded safety to a proof of unbounded safety.

The distinguishing feature of our approach is that it increases the bound for the safety check *exponentially* in the number of iterations, while other approaches do this linearly. That is, in the n^{th} iteration, traditional algorithms check bounded safety up to n steps, but our approach does up to 2^{n+1} steps. However, we do *not* unroll the transition relation an exponential number of times. Instead, we maintain a sequence of transition formulas (i.e., each formula contains only *two* copies of the state variables) where each element over-approximates twice as many steps of transition relation *Tr* as its predecessor. This is the *Transition Power Abstraction* (TPA) sequence. We denote the n^{th} element of the

[2]This condition is important to prevent related SMT queries from growing in complexity in terms of the number of variables.

sequence as $ATr^{\leq n}$ and we require that it *over-approximates* reachability in $\leq 2^n$ steps of Tr, i.e.,

$$Id(X, X') \vee Tr(X, X') \vee Tr^2(X, X') \vee \ldots \vee Tr^{2^n}(X, X') \implies ATr^{\leq n}(X, X'). \quad (4.4)$$

Moreover, we require that $ATr^{\leq 0} \equiv Id \vee Tr$. Thus, $ATr^{\leq 0}$ is *not* an over-approximation but a precise relation capturing true reachability in 0 or 1 steps.

4.3.1 TPA Sequence for Bounded Reachability Queries

The core of our model-checking algorithm lies in answering bounded reachability queries, i.e., whether some *target* states are reachable from some *source* states within some bounded number of steps. The algorithm *uses* the TPA sequence to answer such queries and, at the same time, it *extends* the sequence and *refines* its existing elements.

Intuitively, the sub-procedure works as follows: Given two sets of states, *Source* and *Target*, and n^{th} element of the current TPA sequence $ATr^{\leq n}$, it issues the following SMT query:

$$Sat?[Source(X) \wedge ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X'') \wedge Target(X'')]. \quad (4.5)$$

If this query is unsatisfiable, there is no *intermediate* state reachable from *Source* using one step of $ATr^{\leq n}$ that, at the same time, can reach *Target* in yet another step of $ATr^{\leq n}$. Since one step of $ATr^{\leq n}$ over-approximates reachability in 0 to 2^n steps of Tr, no trace of length $\leq 2^{n+1}$ exists from *Source* to *Target*. Thus, the procedure can immediately conclude that no state from *Target* is reachable from any state in *Source* in $\leq 2^{n+1}$ steps.

Additionally, it is also possible to learn new information about the reachability in $\leq 2^{n+1}$ steps in the form of an interpolant between $ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X'')$ and $Source(X) \wedge Target(X'')$. The properties of interpolation guarantee that the interpolant contains only variables X, X'' (i.e., it does not contain X'), it over-approximates $ATr^{\leq n} \circ ATr^{\leq n}$, and it does not relate any source state with a target state. The relation defined by such an interpolant satisfies the condition from Equation (4.4) for the $n{+}1^{\text{st}}$ element of TPA sequence, and the current TPA sequence can be refined by conjoining the interpolant (after renaming of variables) to its $n{+}1^{\text{st}}$ element.

If query from Equation (4.5) is satisfiable, there exists some *intermediate* state m that can be reached from *Source* by one step of $ATr^{\leq n}$ and can reach *Target* by yet another step of $ATr^{\leq n}$. If $n = 0$, the procedure returns and reports the answer "reachable" as $ATr^{\leq 0}$ is *precise*, not over-approximating. Otherwise, such an intermediate state m can be seen as a potential point on the trace from *Source* to *Target*, and this trace can be shown to be real if there exist two real traces: from *Source* to m and from m to *Target*. The existence of these two real traces can be checked recursively.

The pseudocode for the procedure is given in Algorithm 4.2. We first explain the steps in more detail and demonstrate a run of the algorithm on our example from Section 4.2. Then, we prove the correctness and termination of Algorithm 4.2, from which follow the correctness of Algorithm 4.1 and its termination for unsafe systems.

```
input  : level $n$, source states $Source$, target states $Target$
output : subset of $Target$ reachable from $Source$ within $2^{n+1}$ steps
global : TPA sequence $ATr^{\leq 0}, \dots, ATr^{\leq n}, \dots$
Function IsReachable($n$, $Source$, $Target$):
1     while $true$ do
2         $query \leftarrow Source(X) \wedge ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X'') \wedge Target(X'')$
3         $sat\_res \leftarrow Sat?[query]$
4         if $sat\_res = UNSAT$ then
5             $I \leftarrow Itp(ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X''), Source(X) \wedge Target(X''))$
6             $ATr^{\leq n+1} \leftarrow ATr^{\leq n+1} \wedge I[X'' \mapsto X']$
7             return $\emptyset$
8         else
9             if $n = 0$ then return $QE(\exists X, X'\ query)[X'' \mapsto X]$
10            $Intermediate \leftarrow QE(\exists X, X''\ query)[X' \mapsto X]$
11            $IntermediateReached \leftarrow$ IsReachable($n-1$, $Source$, $Intermediate$)
12            if $IntermediateReached = \emptyset$ then continue
13            $TargetReached \leftarrow$ IsReachable($n-1$, $IntermediateReached$, $Target$)
14            if $TargetReached = \emptyset$ then continue
15            return $TargetReached$
```

Algorithm 4.2. Reachability query using TPA

Bounded Reachability Queries with TPA in Details

Function `IsReachable` takes as input an integer $n \geq 0$, a set of source states, and a set of target states. The output is a subset of target states that are reachable in $\leq 2^{n+1}$ steps of transition relation *Tr*. The output set is empty if no target state is reachable from any source state within the given bound.

The procedure loops until it computes a truly reachable subset of target states or proves all target states unreachable. In each iteration, the procedure uses the current n^{th} element of the TPA sequence. Note that this will be different in each iteration as the TPA sequence will be updated in the recursive calls on lines 12 and 14. The procedure constructs a satisfiability query that represents whether or not an intermediate state is reachable from *Source* using one step of $ATr^{\leq n}$ and, at the same time, can reach *Target* in yet another step of $ATr^{\leq n}$. This query is then passed to a decision procedure for the background theory $\mathcal{T}$ (lines 2 and 3).

Query on line 3 is unsatisfiable. If the query is unsatisfiable, then no target state can be reached from any source state in two steps of $ATr^{\leq n}$. It follows from Equation (4.4) that no target state can be reached from any source state in $\leq 2^{n+1}$ steps. Before indicating the unreachability by returning $\emptyset$ (line 7), the procedure updates the TPA sequence to ensure termination (discussed later): The procedure computes an interpolant between $ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X'')$ and $Source(X) \wedge Target(X'')$ (line 5). After renaming variables, the interpolant is conjoined to the $n{+}1^{\text{st}}$ element of the TPA sequence. In this way, our algorithm is gradually learning new facts about reachability in the system under inspection and refining the abstraction maintained in the TPA sequence. The following example demonstrates this part of the procedure on our motivating example.

Example 4.5. *Consider the system from Figure 4.1 for $N = 3$. This system is unsafe, and the counterexample requires six steps of the transition relation Tr. Here, we focus on the search for counterexample trace and omit the checks for invariant.*

After Algorithm 4.1 initializes the base element of TPA sequence to $(x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x + 1 \wedge y' = ite(x' > 3, y + 1, y))$ it issues a reachability query `IsReachable`$(0, x = 0 \wedge y = 3, x \geq 6 \wedge y = 6)$ in the first iteration of its loop. This translates to a satisfiability check of the formula

$$\begin{aligned}
&x = 0 \wedge y = 3\\
&\wedge ((x' = x \wedge y' = y) \vee (x < 6 \wedge x' = x + 1 \wedge y' = ite(x' > 3, y + 1, y)))\\
&\wedge ((x'' = x' \wedge y'' = y') \vee (x' < 6 \wedge x'' = x' + 1 \wedge y'' = ite(x'' > 3, y' + 1, y')))\\
&\wedge x'' \geq 6 \wedge y'' = 6
\end{aligned}$$

on line 3 of Algorithm 4.2. This query is unsatisfiable, and $x'' \leq x + 2$ is a possible interpolant computed on line 5. After variable renaming, this interpolant refines $ATr^{\leq 1}$, which becomes $x' \leq x + 2$. Then this call to `IsReachable` terminates, and the main loop issues a new reachability query for $n = 1$. This yields a satisfiability query $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \geq 6 \wedge y'' = 6$. Again, this formula is unsatisfiable, and a possible interpolant is $x'' \leq x + 4$. The next element of the TPA sequence, $ATr^{\leq}2$ is refined to $x' \leq x + 4$.

For $n = 2$ (reachability within eight steps), the query on line 3 is satisfiable, and the procedure switches to checking if the counterexample from abstract transition is real or exists only due to a coarse abstraction.

Query on line 3 is satisfiable. If the query on line 8 is satisfiable, a concrete trace of length $\leq 2^{n+1}$ cannot be ruled out at this point. The algorithm proceeds to check the existence of such a trace recursively. In the base case of the recursion, $ATr^{\leq 0}$ is not an over-approximation but a precise relation representing 0 or 1 steps of Tr. Thus, a real trace exists from *Source* to *Target*. The algorithm computes a state formula representing a truly reachable subset of *Target*. First, all except next-next state variables are eliminated from the query by *quantifier elimination* (QE) (line 9). Then, the remaining variables are renamed to state variables.[3]

[3]QE computes maximal reachable subsets. While this is convenient for proving termination of Algorithm 4.2, in practice, quantifier elimination is a costly operation. Our implementation, therefore, supports also the use of *model-based projection* to efficiently under-approximate quantifier elimination (see Section 4.3.4).

If the base case has not been reached yet ($n > 0$), the procedure first computes a set of candidate *intermediate* states by eliminating all except next-state variables from the query (line 10). Then, the procedure recursively calls itself to determine the existence of a trace from *Source* to the newly computed intermediate set with the bound on length halved (line 11). This check has two possible outcomes. If the recursive call returns $\emptyset$, none of the intermediate candidates is reachable (within 2^n steps). Moreover, $ATr^{\leq n}$ must have been strengthened (line 6) before the recursive call returned as to *not* relate any of the source states and intermediate candidates. The procedure then continues to the next iteration (line 12) where it tries to find new intermediate candidates or prove there are none anymore. In case the set returned on line 11 is non-empty, it represents a set of states reachable from *Source* within 2^n steps of *Tr*. The procedure proceeds to check the existence of a trace from these states to the target states (line 13). The reasoning here is the same as for the first recursive call: If *Target* is not reachable, the procedure attempts to find new intermediate candidates in a new iteration. Otherwise, a real trace from *Source* to *Target* exists, and the computed truly reachable states are returned. The returned states are reachable with 2^{n+1} steps as both recursive calls check reachability within 2^n steps.

We continue Example 4.5 to illustrate this phase of Algorithm 4.2.

Example 4.6. *Following Example 4.5, the algorithm is checking bounded reachability between Init and Bad for* $n = 2$*, i.e., within* 8 *steps. The issued satisfiability query is* $x = 0 \wedge y = 3 \wedge x' \leq x + 4 \wedge x'' \leq x' + 4 \wedge x'' \geq 6 \wedge y'' = 6$*. Eliminating all except next-state variables yields* $x' \leq 4 \wedge x' \geq 2$*. This results in the recursive call* `IsReachable`$(1, x = 0 \wedge y = 3, x \leq 4 \wedge x \geq 2)$*. The satisfiability query issued next is* $x = 0 \wedge y = 3 \wedge x' \leq x + 2 \wedge x'' \leq x' + 2 \wedge x'' \leq 4 \wedge x'' \geq 2$*. It is satisfiable and yields* $x' \leq 2 \wedge x' \geq 0$ *after quantifier elimination. Now we reach level* 0 *with a call* `IsReachable`$(0, x = 0 \wedge y = 3, x \leq 2 \wedge x \geq 0)$*. The constructed satisfiability query is again satisfiable, and since we are at level* 0*, the procedure returns a set of states truly reachable from* $x = 0 \wedge y = 3$ *within* 2 *steps. These can be characterized as* $(x = 0 \vee x = 1 \vee x = 2) \wedge y = 3$*. The reachable states are reported to level* 1 *which issues reachability query for the second part:* `IsReachable`$(0, (x = 0 \vee x = 1 \vee x = 2) \wedge y = 3, x \leq 4 \wedge x \geq 0)$*. This is also successful and returns reachable states* $(x = 0 \vee x = 1 \vee x = 2 \vee x = 3 \vee x = 4) \wedge y = 3$*. These are states reachable from Init within* 4 *steps and they are reported to level* 2*. There, the second part of the counterexample is found in a similar way, and the procedure concludes that Bad is truly reachable from Init within* 8 *steps.*

The algorithm's behaviour on these examples can be generalized for the system of Figure 4.1 for larger values of N*. The length of the counterexample is* $2N$ *and let* l *denote* $\lfloor log_2(2N) \rfloor$*. The bounded safety is quickly determined up to* 2^l *steps with* l *calls to* `IsReachable`*, which all return* $\emptyset$ *in their first iteration. On the next iteration, for* $n = l$*,* `IsReachable` *finds the real counterexample, but it requires* $O(2^l)$ *recursive calls to find the counterexample of length in the interval* $(2^l, 2^{l+1}]$*.*

4.3.2 Correctness and Termination

We first prove the correctness and termination of Algorithm 4.2, which then entails the correctness of Algorithm 4.1 and its termination for unsafe systems. We prove the correctness of procedure `IsReachable` separately for the unreachable and the reachable case.

Lemma 4.7. *If* `IsReachable`*(n, Source, Target) returns* $\emptyset$*, then no state from Target can be reached from Source within* 2^{n+1} *steps.*

Proof. The proof relies on the invariant that the sequence $ATr^{\leq 0}, \dots, ATr^{\leq n}, \dots$ maintained by the algorithm satisfies the over-approximating property of the TPA sequence, given in Equation (4.4). This condition holds at the initialization point in Algorithm 4.1. The only update to the elements of the sequence happens in Algorithm 4.2 on line 6. Consider an update on any level $k \leq n$. From the properties of interpolation, we know that $I(X, X'')$ (on line 5) over-approximates $ATr^{\leq k}(X, X') \wedge ATr^{\leq k}(X', X'')$, which represents two steps of the relation $ATr^{\leq k}$. Since $ATr^{\leq k}$ over-approximates $\leq 2^k$ steps of Tr, it follows that $I(X, X'')$ over-approximates $\leq 2^{k+1}$ steps of Tr. Thus, conjoining it to $ATr^{\leq k+1}$ preserves the condition of Equation (4.4).

It follows from Equation (4.4) that when the query on line 3 is unsatisfiable, there exists no trace of length $\leq 2 \times 2^n = 2^{n+1}$ from any source state to any target state. □

Lemma 4.8. *If* `IsReachable`*(n, Source, Target) returns a non-empty set Res, then Res* $\subseteq$ *Target and every state in Res can be reached from some state in Source in* $\leq 2^{n+1}$ *steps.*

Proof. The proof is by induction on n.

Base case: For $n = 0$ $ATr^{\leq 0}$ represents precise reachability in 0 or 1 step. It follows that if the query on line 3 is satisfiable, some target states are truly reachable from the set of source states in ≤ 2 steps. Moreover, the properties of QE guarantee that $Res = QE(\exists X, X' \; query)[X'' \mapsto X]$ is a subset of $Target(X)$ that are reachable from $Source$ using $ATr^{\leq 0} \circ ATr^{\leq 0}$.

Inductive case: Suppose the claim holds for $n - 1$. If at level n the procedure returned a non-empty set, it must have been the case that the first recursive call (line 11) returned a non-empty set *IntermediateReached* of states truly reachable from *Source* in $\leq 2^n$ steps, by our induction hypothesis. Additionally, the second recursive call (line 13) also returned a non-empty set *TargetReached* that, according to our induction hypothesis, is a subset of *Target* truly reachable from *IntermediateReached* in $\leq 2^n$ steps. It follows that *TargetReached* is a subset of *Target* truly reachable from *Source* in $\leq 2^{n+1}$ steps. □

The correctness of procedure `IsReachable` extends naturally to the correctness of our main procedure.

Theorem 4.9 (Correctness)**.** *If Algorithm 4.1 returns UNSAFE, then the system* $\mathcal{S}$ *is unsafe, i.e., some bad state is reachable from some initial state.*

Proof. Algorithm 4.1 returns UNSAFE only if `IsReachable` returns a non-empty set of states for some n. From the correctness of `IsReachable`, it follows that the returned set is a subset of *Bad* reachable from *Init* in $\leq 2^{n+1}$ steps. Thus there exists a counterexample trace in the system. □

Next, we want to show that if there exists a counterexample trace in the system, our procedure will eventually report it. This boils down to the question of termination of a single call to `IsReachable`.

Lemma 4.10. *Assume that the satisfiability check (line 3) terminates, i.e., that the background theory $\mathcal{T}$ is decidable, and that $\mathcal{T}$ has procedures for interpolation and quantifier elimination.*[4] *Then, a single call to `IsReachable` always terminates.*

Proof. The proof proceeds by induction on level n. The base case ($n = 0$) trivially terminates after a single satisfiability query on line 3.

For the inductive case, consider the first iteration of the loop. If the query is unsatisfiable, the procedure terminates. If it is satisfiable, quantifier elimination yields a set of states $\mathit{Intermediate} = \{m \mid \exists s \in \mathit{Source}, \exists t \in \mathit{Target} : (s, m) \in \mathit{ATr}^{\leq n} \wedge (m, t) \in \mathit{ATr}^{\leq n}\}$. Now consider the first recursive call (line 11). By induction, it terminates. If it returns $\emptyset$, then, by properties of the interpolation, $\mathit{ATr}^{\leq n}$ has been strengthened such that $\forall s \in \mathit{Source}, \forall m \in \mathit{Intermediate} : (s, m) \notin \mathit{ATr}^{\leq n}$ now holds. Consequently, in the second iteration, the query on line 3 must be unsatisfiable, and the procedure terminates.

Now consider the situation when the recursive call on line 11 returned a non-empty set *IntermediateReached*. The procedure continues to the second recursive call (line 13), which also terminates, by induction. If the returned set *TargetReached* is non-empty, the procedure terminates (line 15). If it is empty, no state reachable from *Source* in $\leq 2^n$ steps of *Tr* can reach any state in *Target* in another $\leq 2^n$ steps. Moreover, $\mathit{ATr}^{\leq n}$ has been strengthened to not relate any state from *IntermediateReached* with a state in *Target*. In the second iteration, the query on line 3 could still be satisfiable. However, the extracted *Intermediate* (of the second iteration) cannot contain states that are reachable from *Source* in $\leq 2^n$ steps. Thus, the first recursive call (line 11) in the second iteration must return $\emptyset$. This is followed by an unsatisfiable query (line 3) in the third iteration, after which the procedure terminates. □

The immediate consequence of Lemma 4.10 is that our main procedure will find a counterexample if one exists.

Theorem 4.11. *If a counterexample exists in the system, Algorithm 4.1 terminates with the UNSAFE result.*

[4]The linear arithmetic theories satisfy these assumptions.

4.3.3 Proving Safety

Besides the search for counterexample traces with `IsReachable` procedure, Algorithm 4.1 also attempts to prove safety by discovering safe inductive transition invariant in the procedure `HasInvariant`. On one extreme, this procedure could be implemented to always return `False`, which would turn Algorithm 4.1 into a BMC-like algorithm, though with the bound increasing exponentially in the number of iterations. However, it is possible to take advantage of the elements of the TPA sequence, which are guaranteed to satisfy some of the properties of a safe inductive transition invariant. Moreover, the missing property can be checked for with a satisfiability query.

It trivially follows from Equation (4.4) that each element of the TPA sequence satisfies condition 1 of the inductive transition invariant. Moreover, from a certain point onwards, the elements are also safe.

Observation 4.12. *After call to* `IsReachable`$(n, \mathit{Init}, \mathit{Bad})$ *returns* $\emptyset$*,* $\mathit{ATr}^{\leq n+1}$ *is safe for the rest of the run of Algorithm 4.1.*

This follows from the properties of Craig interpolants and the fact that the elements can only be strengthened afterwards. Thus, only the condition for the inductive step of the inductive transition invariant needs to be checked.

Lemma 4.13. *Assume that for some* $n > 0$*,* $\mathit{Init} \lhd \mathit{ATr}^{\leq n} \circ \mathit{Tr} \subseteq \mathit{Init} \lhd \mathit{ATr}^{\leq n}$ *or that* $\mathit{Tr} \circ \mathit{ATr}^{\leq n} \rhd \mathit{Bad} \subseteq \mathit{ATr}^{\leq n} \rhd \mathit{Bad}$ *after* `IsReachable`$(n-1, \mathit{Init}, \mathit{Bad})$ *returned* $\emptyset$*. Then* $\mathit{ATr}^{\leq n}$ *is a safe inductive grounded transition invariant.*

Proof. We have established above that $\mathit{ATr}^{\leq n}$ is safe and satisfies the base condition for inductive transition invariant. The assumption of the lemma is exactly the inductive condition for inductive grounded transition invariant from Definition 4.3. □

A possible implementation of `HasInvariant`, which we use in our implementation, thus consists of checking the condition of Lemma 4.13 for each of the (updated) elements of the TPA sequence. This corresponds to two SMT checks per element. If the condition is satisfied for any element, a safe inductive (grounded) invariant has been discovered. Algorithm 4.1 reports that no counterexample exists and the transition system is safe.

Note that it is sufficient to test for grounded transition invariants; if a transition formula is an inductive transition invariant, it is also a grounded transition invariant.

4.3.4 Under-approximating Quantifier Elimination with Model-based Projection

Model-based projection (MBP) [137] is a recent technique for under-approximating quantifier elimination for existentially quantified formulas. In short, given an existentially quantified formula $\exists X \phi(X, Y)$, MBP is a function that maps each model of ϕ to a quantifier-free formula that implies $\exists X \phi(X, Y)$ and is true in the model. Moreover, it is required that the function has a finite image (it produces only finitely many quantifier-free under-approximations) and the disjunction of the image is equal to the quantified formula. Efficient model-based projections have been discovered for various theories, most notably for linear real and integer arithmetic [33, 137], but also for algebraic datatypes [33], arithmetic signature of bit-vectors [191] and arrays[5] [135].

Quantifier elimination in Algorithm 4.2 can be replaced by MBP in a straightforward way. On line 3, if the query is satisfiable, we obtain from the SMT solver a model witnessing the satisfiability. Then, we replace QE with MBP using the obtained model on lines 9 and 10. It is easy to check that the proof of Lemma 4.8 remains valid with this change, and thus also the result of Theorem 4.9. We demonstrate the practical advantage of MBP over QE experimentally in Section 5.7.1.

4.4 Split Transition Power Abstraction

This section presents a variant of the TPA algorithm that is more tailored to proving safety. The possible implementation of the procedure `HasInvariant` of Algorithm 4.1 proposed earlier uses only induction as a single proof rule, and the only candidates for inductive transition invariants available are the elements of the TPA sequence. Here, we show that it is possible to make the search for transition invariant more powerful; however, the procedure for answering bounded reachability queries gets more complex. First, we obtain an additional source of candidates for transition invariants by *splitting* the TPA sequence. Then, we strengthen the proof rule for proving safety from induction to k-induction. Thus, the algorithm will search not only for inductive but also k-inductive transition invariants.

The splitting, combined with k-induction, results in much better performance in discovering safe transition invariants, even solving benchmarks not solvable by state-of-the-art tools. The experiments and analysis of the results are presented later, in Section 5.7.1.

4.4.1 k-inductive Transition Invariants

k-induction principle [73, 186] generalizes induction and can also be used as a proof rule for safety properties: A state formula $S(X)$ is a *k-inductive invariant* iff

- $\mathit{Init}(X^{(0)}) \wedge \mathit{Tr}^i(X^{(0)}, X^{(i)}) \implies S(X^{(i)})$ for $0 \leq i < k$,
- $\bigwedge_{i=0}^{k-1} S(X^{(i)}) \wedge \mathit{Tr}(X^{(i)}, X^{(i+1)}) \implies S(X^{(k)})$.

If a safe k-inductive invariant exists, the system is safe. Note that the definition of inductive invariant coincides with 1-inductive invariant. k-inductive invariants can be more compact than inductive invariants, and, for some theories, k-induction is strictly stronger than induction [129].

[5]MBP for arrays does not satisfy the finite image condition

The concept of inductive transition invariant can be generalized to k-inductive transition invariant in the same way as for the state version.

Definition 4.14 (k-inductive transition invariant)**.** *A transition formula $T(X, X')$ is a* k-inductive *transition invariant iff the base condition*

$$\mathit{Tr}^i \subseteq T \quad \text{for } 0 \leq i < k, \tag{k-base}$$

and either of the following inductive conditions hold

$$\bigwedge_{i=1}^{k} T(X^{(0)}, X^{(i)}) \wedge \mathit{Tr}(X^{(i)}, X^{(i+1)}) \implies T(X^{(0)}, X^{(k+1)}), \tag{k-ind-fwd}$$

$$\bigwedge_{i=1}^{k} T(X^{(i)}, X^{(k+1)}) \wedge \mathit{Tr}(X^{(i-1)}, X^{(i)}) \implies T(X^{(0)}, X^{(k+1)}). \tag{k-ind-bwd}$$

Observation 4.15. *k-inductive transition invariant is indeed a transition invariant.*

Proof. We need to show that if $(s, t) \in \mathit{Tr}^*$ then $(s, t) \in T$. The proof proceeds by strong induction on the length l of the shortest trace from s to t. The base case covers all $l < k$, in which case $(s, t) \in T$ follows directly from (k-base).

For the induction step, suppose that $l \geq k$ and that the claim holds for all traces of shorter length. We analyze the case of (k-ind-fwd): Take a state m that lies exactly k steps before t on the trace from s. By the induction hypothesis, every state n that lies between m (including) and t (excluding) can be reached from s by one step of T, i.e., $(s, n) \in T$. It follows by (k-ind-fwd) that $(s, t) \in T$. The case of (k-ind-bwd) is analogous; only the k predecessors of t are replaced by k successors of s in the reasoning. □

Similar to inductive transition invariants, it is possible to consider *grounded* version of k-inductive transition invariant.

Definition 4.16 (left- and right-grounded k-inductive transition invariant)**.** *A transition formula $T(X, X')$ is a left-grounded k-inductive transition invariant if the following conditions hold:*

$$\mathit{Init} \lhd \mathit{Tr}^i \subseteq \mathit{Init} \lhd T \quad \text{for } 0 \leq i < k, \tag{left-k-base}$$

$$\mathit{Init}(X^{(0)}) \wedge \bigwedge_{i=1}^{k} T(X^{(0)}, X^{(i)}) \wedge \mathit{Tr}(X^{(i)}, X^{(i+1)}) \implies T(X^{(0)}, X^{(k+1)}). \tag{left-k-ind-fwd}$$

Similarly, the transition formula $T(X, X')$ is a right-grounded k-inductive transition invariant if the following conditions hold:

$$\mathit{Tr}^i \rhd \mathit{Bad} \subseteq T \rhd \mathit{Bad} \quad \text{for } 0 \leq i < k, \tag{right-k-base}$$

$$\bigwedge_{i=1}^{k} T(X^{(0)}, X^{(i)}) \wedge \mathit{Tr}(X^{(i)}, X^{(i+1)}) \wedge \mathit{Bad}(X^{(k+1)}) \implies T(X^{(0)}, X^{(k+1)}). \tag{right-k-ind-bwd}$$

Observation 4.17. *Grounded k-inductive transition invariant is indeed a grounded transition invariant.*

Proof. Same as the proof of Observation 4.15, with restricting s to belong to *Init* for the left-grounded case and restricting t to belong to *Bad* for the right-grounded case. ☐

Note that the stronger version (with weaker antecedent) of (left-k-ind-fwd) $\mathit{Init}(X^{(0)}) \wedge T(X^{(0)}, X^{(1)}) \wedge \mathit{Tr}^k(X^{(1)}, X^{(k+1)}) \implies T(X^{(0)}, X^{(k+1)})$ can be nicely expressed in the set notation as $\mathit{Init} \lhd T \circ \mathit{Tr}^k \subseteq \mathit{Init} \lhd T$. Analogously, the stronger version of (right-k-ind-bwd) can be expressed as $\mathit{Tr}^k \circ T \rhd \mathit{Bad} \subseteq T \rhd \mathit{Bad}$.

4.4.2 Intuition behind Splitting the TPA Sequence

Suppose that we want to improve the ability of Algorithm 4.1 to prove safety, but we want to do as minimal changes as possible. We could try adding checks for k-inductive transition invariant instead of just for inductive transition invariant. However, it is not clear how to do it efficiently. $\mathit{ATr}^{\leq n}$, n^{th} element of the TPA sequence, satisfies the base case (k-base) for k up to 2^n. But checking the inductive step for $k = 2^n$ translates into a satisfiability query with $2^n + 2$ copies of the state variables. Thus the checks would get too complex very quickly for increasing values of n. To remedy the situation, we could apply the same idea that led to the TPA sequence. Instead of checking the implication from (k-ind-fwd), we could consider a stronger implication (with weaker antecedent) $T(X, X') \wedge \mathit{Tr}^k(X', X'') \implies T(X, X'')$. Taking $T := \mathit{ATr}^{\leq n}$ and $k = 2^n$ this amounts to a check of $\mathit{ATr}^{\leq n} \circ \mathit{Tr}^{2^n} \subseteq \mathit{ATr}^{\leq n}$. Unfortunately, this would still require $2^n + 2$ copies of the state variables. Now we could re-use the over-approximating properties of the TPA sequence and further weaken the antecedent, replacing Tr^{2^n} with $\mathit{ATr}^{\leq n}$. However, this over-approximation is too coarse, as the resulting check $\mathit{ATr}^{\leq n} \circ \mathit{ATr}^{\leq n} \subseteq \mathit{ATr}^{\leq n}$ is strictly subsumed by our original check for inductive transition invariant $\mathit{ATr}^{\leq n} \circ \mathit{Tr} \subseteq \mathit{ATr}^{\leq n}$. Thus, this is not the right approach. Nevertheless, it hints at a possible improvement. Instead of replacing Tr^{2^n} with $\mathit{ATr}^{\leq n}$, which captures reachability in *up to* 2^n steps, we could maintain more precise over-approximation that would capture reachability in *exactly* 2^n steps.

We can arrive at the idea of maintaining *another* over-approximating sequence also by identifying a potentially redundant work in the TPA algorithm and trying to resolve the redundancy. Let's revisit the inductive definition of $R^{\leq n}$, given in Equation (4.3). The intuition behind this inductive definition is that every trace of length $\leq 2^{n+1}$ can be obtained as a concatenation of two traces of length $\leq 2^n$. However, there can be multiple ways to decompose such a trace into two smaller traces (see Figure 4.2). Proving one such decomposition infeasible does not entail that others are also infeasible.

The idea of splitting the reachability sequence arises naturally from an attempt to fix this redundancy. The reasoning is as follows: Instead of concatenating two steps of $R^{\leq n}$ to obtain $R^{\leq n+1}$, we replace one of these steps with a step of $R^{=n} = \mathit{Tr}^{2^n}$, which represents reachability in *exactly* 2^n steps. However, $R^{\leq n} \circ R^{=n}$ covers only traces of length from

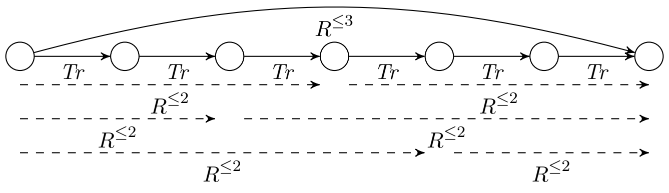


Figure 4.2. Three different ways of decomposing trace of length six into two traces of length at most four

2^n to 2^{n+1}. To keep the smaller lengths covered as well, we can add $R^{\leq n}$. The result, $R^{\leq n+1} = R^{\leq n} \cup R^{\leq n} \circ R^{=n}$, *almost* gives us the unique deconstruction we are seeking. The exceptions are traces of length *exactly* 2^n, which are covered by both $R^{\leq n}$ and $R^{\leq n} \circ R^{=n}$. The final step is a realization that this last redundancy is removed by replacing the relation $R^{\leq n}$ by $R^{<n}$. The sequence $R^{<}$ has the following inductive definition:[6]

$$\begin{aligned} R^{<0} &= Id, \\ R^{<n+1} &= R^{<n} \cup R^{<n} \circ R^{=n}, \end{aligned} \tag{4.6}$$

with the sequence $R^{=}$ also defined inductively:

$$\begin{aligned} R^{=0} &= Tr, \\ R^{=n+1} &= R^{=n} \circ R^{=n}. \end{aligned} \tag{4.7}$$

Notice that we have effectively *split* the $R^{\leq}$ sequence into two sequences $R^{<}$ and $R^{=}$, because $R^{\leq n} = R^{<n} \cup R^{=n}$. Now, decomposing a trace according to the inductive definitions from Equations (4.6) and (4.7) is *unique.* For example, there is only one way to decompose the trace of length six from Figure 4.2, now viewed as one step of $R^{<3}$, according to Equation (4.6): first two steps are covered by $R^{<2}$ and the last four steps are covered by $R^{=2}$.

Following the TPA template, we do not use the sequences $R^{<}$ and $R^{=}$ directly. We build over-approximating sequences $\text{TPA}^{<}$ and $\text{TPA}^{=}$ whose representation in terms of copies of state variables does not blow up with increasing n. The elements of the over-approximating sequences $\text{TPA}^{<}$ and $\text{TPA}^{=}$ are denoted as $ATr^{<n}$ and $ATr^{=n}$, respectively, and we require that

$$ATr^{<n} \supseteq R^{<n} = Id \cup Tr \cup Tr^2 \cup \cdots \cup Tr^{2^n-1}, \tag{4.8}$$

$$ATr^{=n} \supseteq R^{=n} = Tr^{2^n}. \tag{4.9}$$

Next, we will see how splitting the TPA sequence affects the procedures for answering bounded reachability queries and for discovering safe transition invariants.

[6]An alternative inductive definition $R^{<n+1} = R^{<n} \cup R^{=n} \circ R^{<n}$ leads to a different variant of our algorithm.

```
input  : transition system $\mathcal{S} = \langle Init, Tr, Bad \rangle$
global : TPA$^<$ sequence $ATr^{<0}, \dots, ATr^{<n}, \dots$
         TPA$^=$ sequence $ATr^{=0}, \dots, ATr^{=n}, \dots$ (lazily initialized to $true$)
Function IsSafeSplitTPA($\langle Init, Tr, Bad \rangle$):
1   $ATr^{<0} \leftarrow Id$;   $ATr^{=0} \leftarrow Tr$;   $n \leftarrow 0$
2   while $true$ do
3       if IsReachableLt($n, Init, Bad$) $\neq \emptyset$ or IsReachableEq($n, Init, Bad$) $\neq \emptyset$
          then return UNSAFE
4       if HasInvariant($\mathcal{S}, n$) then return SAFE
5       $n \leftarrow n + 1$
```

Algorithm 4.3. SPLIT-TPA's main procedure

4.4.3 Main Procedure

We refer to the version of the TPA algorithm that uses the split sequences as SPLIT-TPA. Its main procedure follows the same template as Algorithm 4.1 and is given in Algorithm 4.3. Next, we present the implementation of the methods `IsReachableLt` and `IsReachableEq` for answering bounded reachability queries and the implementation of the method `HasInvariant` for discovering safe transition invariant.

4.4.4 Bounded Reachability Checks with Split Sequences

Compared to the first algorithm, SPLIT-TPA performs the bounded reachability check at level n in two phases. First, `IsReachableLt` checks all traces of length *strictly smaller* than 2^{n+1}. Then, `IsReachableEq` checks all traces of length *exactly* 2^{n+1}.

Recall that the procedure `IsReachable` from Algorithm 4.2 follows the inductive definition of $R^{\leq}$ (4.3). The procedures `IsReachableEq` and `IsReachableLt` follow in the same way the inductive definitons of $R^{=}$ (4.7) and $R^{<}$ (4.6). Note that the inductive step for $R^{=}$ has the same form as that for $R^{\leq}$. Consequently, `IsReachableEq` is *the same* as `IsReachable`, with the modification that all references to TPA sequence and its elements $ATr^{\leq n}$ are replaced by TPA$^{=}$ sequence and its elements $ATr^{=n}$. For completeness, we give the pseudocode in Algorithm 4.4.

`IsReachableEq` takes as input state formulas representing source and target states and a number n representing the current level. It outputs either a non-empty subset of *Target* that is truly reachable from *Source* in exactly 2^{n+1} steps of *Tr*, or an empty set if no trace from *Source* to *Target* of length 2^{n+1} exists. The procedure `IsReachableLt` is designed to complement `IsReachableEq` by covering all traces with $<2^{n+1}$ steps. It is given in Algorithm 4.5.

Since the code of `IsReachableLt` is more complex, we explain it in more detail. It first assembles the query for an abstract trace (lines 2–4) and sends it to the satisfiability solver (line 5). Following the inductive definition of Equation (4.6), the abstract trace

```
input  : level $n$, source states $Source$, target states $Target$
output : subset of $Target$ reachable from $Source$ in exactly $2^{n+1}$ steps
global : TPA$^=$ sequence $ATr^{=0}, \dots, ATr^{=n}, \dots$
Function IsReachable($n$, $Source$, $Target$):
1    while $true$ do
2        $query \leftarrow Source(X) \wedge ATr^{=n}(X, X') \wedge ATr^{=n}(X', X'') \wedge Target(X'')$
3        $sat\_res \leftarrow Sat?[query]$
4        if $sat\_res = UNSAT$ then
5            $I \leftarrow Itp(ATr^{=n}(X, X') \wedge ATr^{=n}(X', X''), Source(X) \wedge Target(X''))$
6            $ATr^{=n+1} \leftarrow ATr^{=n+1} \wedge I[X'' \mapsto X']$
7            return $\emptyset$
8        else
9            if $n = 0$ then return $QE(\exists X, X'\ query)[X'' \mapsto X]$
10           $Intermediate \leftarrow QE(\exists X, X''\ query)[X' \mapsto X]$
11           $IntermediateReached \leftarrow$ IsReachable($n - 1, Source, Intermediate$)
12           if $IntermediateReached = \emptyset$ then continue
13           $TargetReached \leftarrow$ IsReachable($n - 1, IntermediateReached, Target$)
14           if $TargetReached = \emptyset$ then continue
15           return $TargetReached$
```

Algorithm 4.4. Reachability query using TPA$^=$ sequence

consists of either one step of $ATr^{<n}$ or a step of $ATr^{<n}$ followed by a step of $ATr^{=n}$. If no such abstract trace exists (line 6), the procedure reports that no real trace of length $<2^{n+1}$ exists (line 9). Before reporting the result, it uses Craig interpolation [68] to refine the abstraction at the next level (line 8).

If an abstract trace exists (line 10), the procedure checks whether there is a corresponding real trace. On level 0 (line 11), the discovered abstract trace is real, and the procedure returns a reachable subset of target states. On other levels, the procedure first determines which abstract trace has been found and then tries to refine it.

The first possibility is that the abstract trace is a single step of $ATr^{<n}$ (line 12). The refinement of this single abstract step is checked with a single recursive call. If the refinement is not successful, the procedure attempts to find a new abstract trace (line 14). Otherwise, the reached target states from the recursive call are returned (line 15).

The second possibility is that the abstract trace consists of one step of $ATr^{<n}$ followed by one step of $ATr^{=n}$ (line 16). One after another, the procedure attempts to refine these abstract steps into a real trace by calling the corresponding procedures `IsReachableLt` and `IsReachableEq` with decreased bound. If any of the two steps cannot be refined, that abstract trace has been refuted, and the procedure attempts to find a new abstract trace (lines 19, 21). If both abstract steps have been successfully refined, a reachable subset of target states is reported (line 22).

```
input  : level $n$, source states $Source$, target states $Target$
output : subset of target states truly reachable in $<2^{n+1}$ steps
global : TPA$^<$ sequence $ATr^{<0}, \dots, ATr^{<n}, \dots$,
         TPA$^=$ sequence $ATr^{=0}, \dots, ATr^{=n}, \dots$
Function IsReachable($n$,$Source$,$Target$):
1   while $true$ do
2       $opt1 \leftarrow ATr^{<n}[X' \mapsto X'']$
3       $opt2 \leftarrow ATr^{<n}(X, X') \land ATr^{=n}(X', X'')$
4       $query \leftarrow Source(X) \land (opt1 \lor opt2) \land Target(X'')$
5       $sat\_res, model \leftarrow Sat?[query]$
6       if $sat\_res = UNSAT$ then
7           $Itp(X, X'') \leftarrow$ GetItp($opt1 \lor opt2, Source(X) \land Target(X'')$)
8           $ATr^{<n+1} \leftarrow ATr^{<n+1} \land Itp[X'' \mapsto X']$
9           return $\emptyset$
10      else
11          if $n = 0$ then return $QE(\exists X, X' : query)[X'' \mapsto X]$
12          if $model \models opt1$ then
13              $TargetReached \leftarrow$ IsReachableLt($n-1, Source, Target$)
14              if $TargetReached = \emptyset$ then continue
15              return $TargetReached$
16          else
17              $Intermediate \leftarrow QE(\exists X, X'' :$
                  $Source(X) \land opt2 \land Target(X''))[X' \mapsto X]$
18              $IntermediateReached \leftarrow$
                  IsReachableLt($n-1, Source, Intermediate$)
19              if $IntermediateReached = \emptyset$ then continue
20              $TargetReached \leftarrow$
                  IsReachableEq($n-1, IntermediateReached, Target$)
21              if $TargetReached = \emptyset$ then continue
22              return $TargetReached$
```

Algorithm 4.5. Reachability query using TPA$^<$ sequence

Similarly to Algorithm 4.2, quantifier elimination can be under-approximated with model-based projection, and we do so in our implementation.

The correctness of the reachability procedures guarantees the correctness of the UNSAFE answer of SPLIT-TPA.

Lemma 4.18. *If* `IsReachableEq`*(n, Source, Target) or* `IsReachableLt`*(n, Source, Target) returns a non-empty set Res, then Res $\subseteq$ Target and every state in Res can be reached from some state in Source in exactly 2^{n+1} steps (for* `IsReachableEq`*) or in $<2^{n+1}$ steps (for* `IsReachableLt`*).*

Proof. By induction on n, relying on the properties of quantifier elimination (QE) and the fact that $ATr^{<0} = Id$ and $ATr^{=0} = Tr$ represent true reachability. □

Theorem 4.19. *If SPLIT-TPA (Algorithm 4.3) returns UNSAFE, then there exists a counterexample trace in the system, i.e., some bad state is reachable from some initial state.*

Proof. Follows directly from Lemma 4.18. □

4.4.5 Discovering Safe Transition Invariants in split-TPA

Similar to Algorithm 4.1, Algorithm 4.3 can prove the given system safe by discovering a safe transition invariant. However, the split sequences TPA$^{<}$ and TPA$^{=}$ provide a richer pool of candidates and allow easy use of k-induction as a proof rule instead of just simple induction. Note that elements $ATr^{<n}$ and $ATr^{=n}$ are guaranteed to be safe if `IsReachableLt`$(n-1, Init, Bad)$ and `IsReachableEq`$(n-1, Init, Bad)$ return $\emptyset$, respectively. We show that it is possible to use satisfiability checks that do not grow in complexity (in terms of the number of variables) to test the elements for the properties of k-inductive transition invariants.

Lemma 4.20. *Assume that for some n, $Init \lhd ATr^{<n} \circ ATr^{=m} \subseteq Init \lhd ATr^{<n}$ for some $0 \leq m \leq n$. Then $ATr^{<n}$ is a left-grounded 2^m-inductive transition invariant.*

If $ATr^{=m} \circ ATr^{<n} \rhd Bad \subseteq ATr^{<n} \rhd Bad$ for some $0 \leq m \leq n$, then $ATr^{<n}$ is a right-grounded 2^m-inductive transition invariant.

Proof. We prove only the first statement; the second one is analogous. We show that $ATr^{<n}$ satisfies (left-k-base) and (left-k-ind-fwd). The base condition (left-k-base) follows trivially from Equation (4.8) and the fact $m \leq n$. The inductive condition (left-k-ind-fwd) is satisfied because $Init \lhd ATr^{<n} \circ ATr^{=m} \subseteq Init \lhd ATr^{<n}$ implies $Init \lhd ATr^{<n} \circ Tr^{2^m} \subseteq Init \lhd ATr^{<n}$ which is a stronger version of (left-k-ind-fwd). □

Note that the case of $m = 0$ in Lemma 4.20 corresponds to Lemma 4.13 as $ATr^{=0} \equiv Tr$. If full transition invariants are required, we can instead check for the stronger condition:

Observation 4.21. *If $ATr^{<n} \circ ATr^{=m} \subseteq ATr^{<n}$ or $ATr^{=m} \circ ATr^{<n} \subseteq ATr^{<n}$ for some $0 \leq m \leq n$ then $ATr^{<n}$ is 2^m-inductive transition invariant.*

Next, we show that checking a fixed-point condition on the elements of TPA$^{=}$ can also yield a transition invariant.

Lemma 4.22. *Assume that for some n, $Init \lhd ATr^{<n} \circ ATr^{=n} \circ ATr^{=n} \subseteq Init \lhd ATr^{<n} \circ ATr^{=n}$ then $Init \lhd ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ is a left-grounded 2^n-inductive transition invariant. If $ATr^{=n} \circ ATr^{=n} \circ ATr^{<n} \rhd Bad \subseteq ATr^{=n} \circ ATr^{<n} \rhd Bad$ then $ATr^{<n} \cup ATr^{=n} \circ ATr^{<n} \rhd Bad$ is a right-grounded 2^n-inductive transition invariant.*

Proof. The proof uses the same ideas as the proof of Lemma 4.20. Again, we show the proof only for the first claim. The base case (left-k-base) is satisfied by the first component $ATr^{<n}$ of the formula. For the inductive case (left-k-ind-fwd) we show that even stronger condition holds: $Init \triangleleft ATr^{<n} \cup ATr^{<n} \circ ATr^{=n} \circ ATr^{=n} \subseteq Init \triangleleft ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$. Suppose that $(s,t) \in (ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}) \circ ATr^{=n}$ and $s \in Init$. Then there is m such that $(s,m) \in ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ and $(m,t) \in ATr^{=n}$. There are two possibilities:

- $(s,m) \in ATr^{<n}$: It follows that $(s,t) \in ATr^{<n} \circ ATr^{=n}$.
- $(s,m) \in ATr^{<n} \circ ATr^{=n}$: It follows that $(s,t) \in ATr^{<n} \circ ATr^{=n} \circ ATr^{=n}$. But then $(s,t) \in ATr^{<n} \circ ATr^{=n}$ by the assumption of the lemma.

□

Similarly to Lemma 4.20, full transition invariants can be discovered by checking a stronger condition:

Observation 4.23. *If $ATr^{=n} \circ ATr^{=n} \subseteq ATr^{=n}$ then both $ATr^{<n} \cup ATr^{<n} \circ ATr^{=n}$ and $ATr^{<n} \cup ATr^{=n} \circ ATr^{<n}$ are 2^n-inductive transition invariants.*

One difference compared to proving some $ATr^{<n}$ a transition invariant is that the second component of the candidate formulas from Lemma 4.22, $ATr^{<n} \circ ATr^{=n}$ or $ATr^{=n} \circ ATr^{<n}$, is not necessarily safe. However, this can be easily checked by yet another satisfiability query.

The procedure `HasInvariant`$(\mathcal{S}, n)$ thus checks the conditions of Lemma 4.20 and Lemma 4.22 for elements of the TPA$^=$ and TPA$^<$ sequences up to $ATr^{<n+1}$ and $ATr^{=n+1}$. These are already safe after `IsReachableLt`$(n, Init, Bad)$ and `IsReachableEq`$(n, Init, Bad)$ returned $\emptyset$. The conditions are checked by issuing a series of satisfiability queries. For example, the condition $ATr^{=n} \circ ATr^{=n} \circ ATr^{<n} \triangleright Bad \subseteq ATr^{=n} \circ ATr^{<n} \triangleright Bad$ holds if and only if the formula $ATr^{=n}(X, X') \wedge ATr^{=n}(X', X'') \wedge ATr^{<n}(X'', X''') \wedge Bad(X''') \wedge \neg ATr^{=n}(X, X'')$ is *unsatisfiable*. Note that each query uses at most 4 copies of the state variables, no matter how large n is. This is achieved by using the over-approximating elements of the TPA$^=$ sequence, instead of 2^n copies of the transition relation Tr.

When the procedure `HasInvariant` returns `true`, it means a safe transition invariant has been discovered. As a consequence, there is no counterexample trace in the system from *Init* to *Bad*, and Algorithm 4.3 returns SAFE. The correctness of this answer is guaranteed by Lemma 4.20 and Lemma 4.22.

Theorem 4.24 (Correctness of SPLIT-TPA)**.** *If SPLIT-TPA returns SAFE, there is no counterexample trace from Init to Bad in $\mathcal{S}$.*

Example 4.25. *As a short example, we consider the analysis of the following multi-phase loop by the algorithm SPLIT-TPA:*

```
v = 0; w = 0;
assume(x > z);
while (v < 1000) {
    if (x < z) v = v + 1;
    else w = w + 1;
    x = x + 1; z = z + 2;
}
assert(w > 0);
```

First, SPLIT-*TPA refutes the existence of a counterexample trace of length less than two, because the assertion after the loop cannot be reached after less than two iterations due to the loop condition. It learns that* $ATr^{<1} \equiv v' \leq v + 1$*. Then it refutes the existence of a counterexample trace of length* 2*, not because of the loop variable, but because of the interaction between variables* x*,* z *and* w*. Namely, it learns that* $ATr^{=1} \equiv x > z \rightarrow w' \geq w + 2$*. In the next iteration of the main loop, when searching for counterexamples of length up to* 4*,* $ATr^{=1}$ *is strengthened with the facts* $x \geq z \rightarrow w' \geq w + 1$ *and* $x < z \rightarrow w' \geq w$*. These three facts together concisely overapproximate the change to* w *after precisely two iterations of the loop. Moreover,* $ATr^{=1}$ *with these three components is closed under composition, i.e.,* $ATr^{=1} \circ ATr^{=1} \subseteq ATr^{=1}$*. Thus,* SPLIT-*TPA already at this point discovers* 2*-inductive transition invariant (based on Observation 4.23), which is also safe. The transition invariant, using* $\vec{a} = (x, z, v, w)$*, is then* $ATr^{<1}(\vec{a}, \vec{a}'') \vee (\exists \vec{a}' : ATr^{<1}(\vec{a}, \vec{a}') \wedge ATr^{=1}(\vec{a}', \vec{a}''))$*, where*

$$ATr^{<1}(\vec{a}, \vec{a}') \equiv w' \geq w \wedge v' \leq v + 1 \wedge ((x' \geq x \wedge z' \leq z) \vee (x' \geq x + 1 \wedge z' \leq z + 2)),$$

$$ATr^{=1}(\vec{a}, \vec{a}') \equiv x > z \rightarrow w' \geq w + 2 \wedge x \geq z \rightarrow w' \geq w + 1 \wedge x < z \rightarrow w' \geq w.$$

Note that the exact value of $ATr^{<1}$ *is not essential in this case, as long as it overapproximates all traces of length less than two and both* $ATr^{<1}$ *and* $ATr^{<1} \circ ATr^{=1}$ *are safe.*

4.5 Alternatives, Extensions and Future Work

The presented algorithms TPA and SPLIT-TPA represent a direct application of the main idea behind transition power abstraction. However, there are variations and extensions yet to be explored. One variation considers the base of the power in TPA. There is no theoretical obstacle to using a different base, for example, three. The n^{th} element $ATr^{\leq n}$ would then summarize 3^n steps of the transition relation Tr instead of 2^n. Evaluation of the advantages and disadvantages of such a change requires further research.

Another variation of the algorithm is related to how the algorithm searches for the counterexample path. Recall the inductive definition of $R^{<}$ (4.6). The inductive step of the definition could be changed to $R^{<n+1} = R^{<n} \cup R^{=n} \circ R^{<n}$. Though this would still define the same relation, it would change Algorithm 4.5. The order of the two abstract steps in the second part of the query (line 3) would be switched to $ATr^{=n}(X, X') \wedge$

$ATr^{<n}(X', X'')$. Consequently, the recursive calls in the refinement of the two-step abstract path (lines 18 and 20) would also be called in reverse order: `IsReachableEq` would refine the step from the source to the intermediate states, and `IsReachableLt` would refine the step from the reached intermediate states to the target states.

The search for the counterexample path can also be altered differently. In the presented version, the real path is built from the source states towards the target states. However, this can also be reversed. Building the real path from the target states towards the source states is also possible. This "backward" flow would be more similar to how PDR-based algorithms propagate proof obligations from the bad states towards the initial states. The combination of the forward and backward analyses has a potential for a great improvement [184] and is an intriguing future work.

The focus of the TPA-based algorithms on *transitions* rather than *states* opens up possibilities for modular analysis of more complex systems, such as general systems of constrained Horn clauses. The information learnt about transitions is not invalidated even when the initial conditions change; this allows efficient independent analysis of multiple connected loops that exchange information about reachable and unreachable states. Our implementation already supports the first step in this direction: analysis of chains of transition systems. The details of this implementation are discussed later, in Section 5.5.5. We imagine the extension to nonlinear systems of Horn clauses as an *unbounded* Spacer-like search over a network of transition systems. The proper design and implementation of this extension is future work.

4.6 Related Work

Many model-checking algorithms search for a safe inductive invariant to prove safety. Candidates for inductive invariants are typically obtained from proofs of bounded safety. The algorithms try to construct the safe inductive invariant either in monolithic [155, 157, 176] or incremental way [41, 58, 85, 113, 137]. Our work follows a similar strategy, but it primarily computes *transition* invariants, not state invariants.

Transition invariants have been introduced in [164] as a proof rule for program verification, especially termination and other liveness properties. Transition predicate abstraction [165] has been introduced as a way to compute transition invariants. Transition invariants have also been used for loop summarization with the goal of proving termination [141]. However, in that work candidates for transition invariants were obtained from specialized abstract domains using techniques from abstract interpretation [67]. In contrast, we use transition invariants to prove safety, with candidates automatically obtained from proofs of bounded safety using Craig interpolation.

Craig interpolation [68] is a popular abstraction technique widely used in model checking. We use standard algorithms to compute interpolants from proofs of unsatisfiability [37, 61, 156]. The integration of domain-specific knowledge [146] is future work.

While in most model checking algorithms interpolants are used as over-approximations of *states*, we use them to over-approximate *transitions*. The idea of abstracting transition relation with interpolants originates from [127]. However, they maintained an abstraction of only a single step of the transition relation. We build two sequences of relations over-approximating doubling number of steps of the transition relation, which are useful both for detecting deep counterexamples and as a source of candidates for safe transition invariant.

Loop acceleration [12, 40, 98] is a loop analysis technique that can prove safety and detect deep counterexamples. It transforms the loop to a single quantifier-free formula representing all possible executions of the loop. While offering significant improvement for a limited types of integer loops, it is not applicable for code with control-flow divergence and/or data structures. Acceleration have also been successfully integrated into interpolation-based model checking [49, 115] where interpolants computed from accelerated paths lead to much better abstraction refinement in the traditional CEGAR algorithm [63]. In contrast, our technique does not accelerate paths but builds over-approximations of bounded number of iterations. It also computes transition interpolants, instead of state interpolants. It is not restricted to any specific type of loops, and it works over any theory supporting interpolation and quantifier elimination.

A speciliazed technique technique for fast detection of deep counterexamples for C programs was proposed in [140]. Given a path through a loop, it computes a new path that *under-approximates* an arbitrary number of iterations of the original path. In contrast to loop acceleration, this technique only under-approximates the loop behaviour. On the other hand, it can handle conditionals and richer background theories. Our technique also focuses on the detection of deep counterexamples, but it is *over-approximating*, which also allows for detecting transition invariants and proving safety. Their prototype aims at C programs only (and does not seem to be maintained anymore). Our implementation works on transition systems in the form of constrained Horn clauses (CHC) and thus is agnostic to the programming language.

The k-induction principle [73, 186] has been successfully used as a replacement for basic inductive reasoning in IC3-style algorithms [106, 129, 193]. k-inductive invariants can be more compact than inductive invariants and for some theories k-induction is a strictly stronger proof rule [129]. While the first, simpler version of our algorithm could only use inductive reasoning for discovering transition invariants, the second version, SPLIT-TPA, uses k-inductive reasoning. We believe that SPLIT-TPA's success on challenging systems can be in large part attributed to the inclusion of k-inductive reasoning, which is missing from our first TPA algorithm.

4.7 Conclusion

In this chapter, we have introduced a novel model-checking algorithm for safety properties of transition systems with a focus on finding deep counterexamples. The core component of the algorithm is a sequence of transition formulas, called *transition power abstraction* (TPA) sequence, where each element over-approximates a sequence of transition steps *twice as long as* its predecessor. TPA sequence allows the algorithm to quickly answer bounded reachability queries, in ideal case *doubling* the size of the covered state space with a *single*, relatively simple SMT query. Additionally, the elements of the TPA sequence serve as automatically discovered candidates for safe transition invariants.

The basic version of the algorithm, with a single TPA sequence, is able to detect *an order of magnitude* longer counterexamples that state-of-the-art model checking algorithms within the same time contraints. However, its ability to prove system safe is very limited. For this reason we introduced an improved version of the algorithm obtained by *splitting* the TPA sequence into two separate sequences. The key ingredients of SPLIT-TPA are more candidates for safe transition invariants and the ability to efficiently check for k-inductive transition invariants, not only basic inductive transition invariants. SPLIT-TPA not only retains the ability to detect deep counterexample, but also significantly improves the ability to prove unbounded safety.

Chapter 5

The Golem Horn Solver

This chapter describes Golem, a solver for the satisfiability of constrained Horn clauses (CHC) developed to support the research reported in this thesis. Golem is an open-source program, written in C++17, and available at GitHub[1]. As input, it takes a system of constrained Horn clauses in the format prescribed by the international competition of Horn solvers (CHC-COMP).[2] The goal is to decide the satisfiability of the input system (see Section 2.4 for details). If there exists an interpretation of the uninterpreted predicates (a model) that satisfies all the clauses, then the system is satisfiable. If a derivation of *false* from the system of clauses exists, then it constitutes proof that the system is unsatisfiable. Correspondingly, Golem outputs `SAT` when it finds a model or `UNSAT` when it finds a proof of unsatisfiability. This output follows the usual semantics of SMT solvers prescribed by SMT-LIB. CHC-COMP also uses this semantics. Optionally, Golem can also output the model or the proof found.

Golem was designed to enable a controlled interaction between SMT-based model-checking algorithms and the SMT solver used for various computational tasks. In particular, the interpolating procedures of the SMT solver were designed to be easily accessible and adaptable to the needs of the verifier engine. The open and modular infrastructure of Golem allowed not only implementing state-of-the-art algorithms such as BMC, k-induction, Impact and Spacer, but also rapidly prototyping our novel TPA-based algorithms (Chapter 4). One of the key design choices was to tightly integrate Golem with the underlying SMT solver OpenSMT. The practical consequences of the tight integration are twofold: It enables fine-tuned use of the SMT solver, its decision and interpolation procedures. Additionally, it enables the re-use of existing data structures for term representation and manipulation, which saves development time and avoids the runtime overhead of a translation between different term representations. During the development of Golem, the integrated architecture allowed tuning of the overall infrastructure and removing potential performance bottlenecks, both in Golem and in

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

[2] `https://chc-comp.github.io/format.html`

OpenSMT. The drawback of this design choice is that it is not possible to easily replace the underlying SMT solver or use multiple different SMT solvers. Other state-of-the-art Horn solvers are integrated quite tightly with their underlying satisfiability engines as well. Spacer [137] lives as a fixed-point engine inside the Z3 solver [72] and, as such, has unrestricted access to its various utilities, including the core SMT solver. Similarly, the interpolating theorem prover Princess [173] is integrated into Eldarica [117].

An official publication for Golem is currently being prepared for submission. However, it was already used in the experiments for evaluating our TPA algorithms in [35, 36]. In the experiments, the implementation of TPA algorithms in Golem was compared to the existing model-checking tools on a set of challenging benchmarks representing multi-phase loops. Additionally, it participated in CHC-COMP in 2021 and 2022; in the 2022 edition, it beat all other Horn solvers except the non-competing Spacer in the LRA-TS, LIA-Lin and LIA-Nonlin tracks. A short description of Golem has been published in the CHC-COMP-21 competition report [95].

The organization of this chapter is the following: First, we introduce terminology regarding the internal representation of a CHC system in Golem. Then we describe the high-level architecture of Golem and focus more on the CHC transformations and reasoning engines it implements. We also briefly discuss some interesting details from our work on OpenSMT as the core component of Golem. After the tool's description, we present results from several experiments. Finally, we discuss possible future directions for Golem and related work.

5.1 Terminology

The terminology for Horn clauses has been introduced in Section 2.4. Golem uses graph representation for a system of Horn clauses. Nodes in the graph correspond to the uninterpreted predicates of the system; edges correspond to the clauses of the system. Additionally, there are two particular nodes *true* and *false*, where *true* is understood as the body predicate of *facts*, and *false* is the head predicate of *queries.* The graph has the following properties:

- It is *labeled.* Each edge is labeled with the *constraint* of the clause represented by the edge.
- It is possibly a *hypergraph.* Each edge connects all the body's predicates to the head's predicate.
- It is *directed.* Each edge is oriented *from* the predicates of the body (*source* nodes) *to* the predicate of the head (*target* node).
- It is possibly a *multigraph.* There can be multiple clauses with the same body and head predicates. Each clause has a distinct edge in the graph, labeled with the clause's constraint.

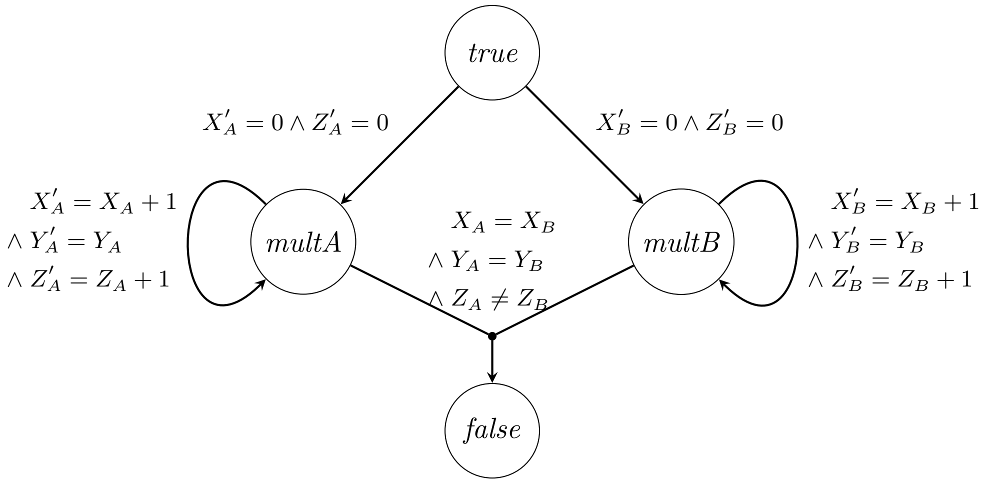


Figure 5.1. Graph representation of the CHC system from Example 5.1

Note that only a clause with multiple predicates in the body is represented by a *hyperedge*, i.e., an edge with multiple source nodes. If the CHC system is *linear* (each clause has at most one predicate in the body), then all edges are standard directed edges, with one source node and one target node. In a standard graph (not hypergraph), we denote a directed edge e as an ordered pair (s,t) where s is the source node of e and t is its target node. We also use helper functions *src* and *tgt* where $src(e) = s$ and $tgt(e) = t$. For a node v we denote $in(v) = \{e \mid tgt(e) = v\}$ the set of v's *incoming* edges and $out(v) = \{e \mid src(e) = v\}$ the set of v's *outgoing* edges. For an edge e, we denote its label, the constraint of the corresponding clause, as $\mathcal{L}(e)$.

Example 5.1. *To illustrate the graph representation, consider the normalized version of the CHC system from Example 2.2 in Section 2.4.*

$$\begin{aligned}
\mathit{true} \land X'_A = 0 \land Z'_A = 0 &\implies \mathit{multA}(X'_A,Y'_A,Z'_A)\\
\mathit{multA}(X_A,Y_A,Z_A) \land X'_A = X_A + 1 \land Z'_A = Z_A + Y_A \land Y'_A = Y_A &\implies \mathit{multA}(X'_A,Y'_A,Z'_A)\\
\mathit{true} \land X'_B = 0 \land Z'_B = 0 &\implies \mathit{multB}(X'_B,Y'_B,Z'_B)\\
\mathit{multB}(X_B,Y_B,Z_B) \land X'_B = X_B + 1 \land Z'_B = Z_B + Y_B \land Y'_B = Y_B &\implies \mathit{multB}(X'_B,Y'_B,Z'_B)\\
\mathit{multA}(X_A,Y_A,Z_A) \land \mathit{multB}(X_B,Y_B,Z_B) \land X_A = X_B \land Y_A = Y_B \land Z_A \neq Z_B &\implies \mathit{false}
\end{aligned}$$

Compared to the original system, we assigned unique variables to each predicate and rewrote the clauses with predicates in the heads using the primed (next-state) versions of the variables. This rewriting is similar to what Golem *uses internally.*[3] *We also explicitly added the predicate true to the facts of the system.*

[3]The real representation in Golem is more complicated because it needs to take into account possibly multiple occurrences of the same predicate in the body of the same clause.

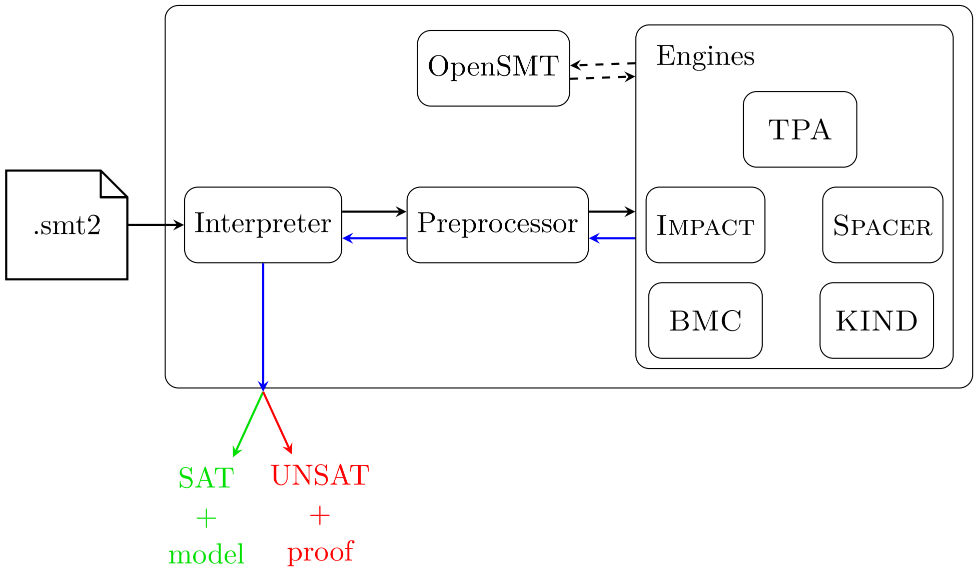


Figure 5.2. High-level architecture of GOLEM

The graph representation of this system is depicted in Figure 5.1. The single incoming edge of false is a hyperedge because it has two source nodes, multA and multB. Its label is the constraint of the corresponding clause, i.e., $X_A = X_B \wedge Y_A = Y_B \wedge Z_A \neq Z_B$.

As described in Section 2.5, the safety of transition systems can be encoded into a CHC representation of a particular form. In the graph representation of a transition system, there is a single node v, beside the special nodes *true* and *false*. Moreover, there are only three edges: from *true* to v, from v to *false* and a self-loop on v. Self-loop on v represents the transition relation, the incoming edge defines the initial states, and the outgoing edge defines the error states. If this CHC system is satisfiable, the interpretation of v's uninterpreted predicate represents a *safe inductive invariant* of the transition system.

5.2 Architecture

The flow of data inside GOLEM is depicted in Figure 5.2. The system of constrained Horn clauses (CHCs) is read from `.smt2` file, a script in an extension of the language of SMT-LIB. `Interpreter` is responsible for interpreting the SMT-LIB script and building the internal representation of the system of CHCs. In GOLEM, CHCs are first *normalized*, and then the system is turned into the graph representation described in Section 5.1. Normalization ensures that unique variables are used to obtain a canonical representation of each predicate. With the canonical representation, clauses are rewritten such that predicates contain only variables (no complex terms) as arguments.

The graph representation of the system is then passed to the `Preprocessor`, which applies various graph transformations. The goal of these transformations is to simplify the graph and make it easier to solve by the chosen back-end engine. The transformations implemented in Golem are described in detail in Section 5.4.

After the preprocessing phase, the transformed graph is sent to the chosen back-end engine. Engines in Golem implement various algorithms for solving the CHC satisfiability problem. Golem contains the implementation of our TPA-based algorithms (Chapter 4), but also re-implementation of state-of-the-art algorithms BMC [29], KIND [186], Impact [157], and Spacer [137]. Users pick the engine to run using a command-line option `--engine`. A detailed description of the engines is given in Section 5.5.

Each engine in Golem implements an SMT-based algorithm; thus, they all rely on an SMT solver for answering SMT queries and providing models and interpolants. Golem relies entirely on OpenSMT [122] for these tasks. Additionally, Golem re-uses the data structures of OpenSMT for representing and manipulating terms.

If an engine solves the CHC system, it reports the satisfiability result. If required, it also reports the witness for its answer: a model for the uninterpreted predicates or a proof of unsatisfiability. Note, however, that this would be a witness for the *transformed* graph, the result of the preprocessing run. Thus, to obtain an accurate witness for the input system, `Preprocessor` must be able to *backtranslate* the witness through the transformations applied to the graph.[4] Only after this backtranslation the answer and the witness are reported to the user.

5.3 Witness Format

Witness for the satisfiability of a CHC system is a model—an interpretation for each uninterpreted predicate that makes all clauses valid. Internally, Golem represents the interpretations as formulas in the background theory, using only the variables of the (normalized) uninterpreted predicate. All interpretations are kept in a single map that maps uninterpreted predicate symbols to their interpretations.[5] The model is presented to the user in the format defined by SMT-LIB [16]: a sequence of SMT-LIB's `define-fun` commands, one for each uninterpreted predicate.

Witness for unsatisfiability of a CHC system is a derivation of *false*. As it stands now, th research community has yet to agree on the exact output format for UNSAT witnesses. Due to its simplicity, Golem follows the trace format used by the Eldarica solver. Internally, Golem stores the derivation as a sequence of derivation steps. Every derivation step is a ground instance of some clause from the system. The ground instances of predicates from the body form the *premises* of the step, and the ground instance of the head's predicate forms the *conclusion* of the step. For the derivation to be valid,

[4]As the graph is translated moving forward through transformation passes, the witness must be translated moving backwards through the same sequence of transformations.

[5]This corresponds to the *relation symbol assignment* for *syntactic* solvability of [176].

the premises of each step must have been derived earlier, i.e., each premise must be a conclusion of some derivation step earlier in the sequence. To the user, the proof is presented as a sequence of derivations of ground instances of the predicates, where each step is annotated with the indices of its premises.

Example 5.2. *Consider the following CHC system and the proof of its unsatisfiability.*

$$\begin{aligned} x > 0 &\implies L1(x) \\ x' = x + 1 &\implies D(x, x') \\ L1(x) \wedge D(x, x') &\implies L2(x') \\ L2(x) \wedge x \leq 2 &\implies \mathit{false} \end{aligned} \qquad \begin{array}{lll} 1. & L_1(1) & \\ 2. & D(1,2) & \\ 3. & L_2(2) & ;1,2 \\ 4. & \mathit{false} & ;3 \end{array}$$

The derivation of false consists of four derivation steps. Step 1 *instantiates the first clause for* $x := 1$*. Step* 2 *instantiates the second clause for* $x := 1$ *and* $x' := 2$*. Step* 3 *applies resolution to the instance of the third clause for* $x := 1$ *and* $x' := 2$ *and facts derived in steps* 1 *and* 2*. Finally, step* 4 *applies resolution to the instance of the fourth clause for* $x := 2$ *and the fact derived in step* 3*.*

5.4 Preprocessing of CHC Systems

Transformations of CHC systems play an important role in the performance of Horn solvers; for example, both Eldarica and Z3 apply many powerful transformations in the preprocessing phase. The goal of transformations is typically to simplify the problem before it is passed to the actual solving algorithm. Some solving techniques apply only to CHC systems of a particular form, and transformations are applied to make the technique applicable. In state-of-the-art Horn solvers, transformations are applied mainly as preprocessing, but they can constitute a solving method on their own. An overview of CHC transformations can be found in [32]. Here we describe only the transformations implemented in Golem, together with the backtranslations for models (witnesses of satisfiability) and proofs (witnesses of unsatisfiability).

Simple-chain summarization

The goal of this transformation is to simplify the CHC graph by replacing a (possibly long) path with a single edge, eliminating the intermediate nodes in the process. A simple chain in the CHC graph is a path between two nodes containing only normal (not hyper) edges such that each intermediate node does not have any other edges except the one incoming and the one outgoing edge present in this path. Each simple chain is replaced by a single edge from the first to the last node. The label of the new edge is a conjunction of all the constraints on the path after an appropriate renaming of the variables. After the addition of the summarizing edge, all the intermediate nodes are removed.

To avoid quantifiers when backtranslating a model, GOLEM relies on interpolation. Suppose that the transformation replaced a path with source node s, target node t and n intermediate nodes $i_1, \ldots, i_n$ with a new edge (s, t). Let $\mathcal{M}$ denote the model for the transformed system. Then the following formula is unsatisfiable:

$$\mathcal{M}(s) \wedge \mathcal{L}(s, i_1) \wedge \bigwedge_{j=1}^{n-1} \mathcal{L}(i_j, i_{j+1}) \wedge \mathcal{L}(i_n, t) \wedge \neg\mathcal{M}(t)$$

The unsatisfiability follows from the definition of the model and the fact that $\mathcal{L}s, t \equiv \mathcal{L}(s, i_1) \wedge \bigwedge_{j=1}^{n-1} \mathcal{L}(i_j, i_{j+1}) \wedge \mathcal{L}(i_n, t)$. The interpretation for the intermediate nodes $i_1, \ldots, i_n$ can be obtained by computing a *path interpolant* for the formula. To perform the backtranslation efficiently, GOLEM utilizes the abilities of OPENSMT to compute path interpolants from a single proof of unsatisfiability.

The backtranslation of a witness of unsatisfiability is also possible. A single derivation step corresponding to the summarizing edge is replaced by a sequence of steps corresponding to the original chain. The values for the ground instances of the intermediate nodes can be computed from the model of the path constraints, where the values for the source and target variables are taken from the ground instances of the derived fact and the premise of the derivation step.

Merging multiple edges with the same source and target

This relatively simple transformation aims to simplify the graph by reducing the number of edges. It combines multiple edges with the same source and target nodes into a single edge. The constraint on the new edge is a disjunction of the replaced edges' constraints. After adding the new edge, the previous edges are removed from the graph. We are effectively shifting work from the reachability algorithm to the underlying SMT solver by applying this transformation. Instead of the reachability algorithm trying to figure out which edge can extend a feasible path, only a single edge is possible. However, the SMT solver needs to determine which disjuncts (if any) can be used. This transformation is currently implemented only for edges with a single source node, i.e., not proper hyperedges.

The backtranslation of a model does not require any change in the model. The backtranslation of an unsatisfiability proof requires only changes in the bookkeeping information, not in the derived facts. For the derivation step that uses the new edge, it needs to choose one of the original edges that also enables the derivation. The right edge can be found by checking which disjunct is satisfied in the satisfiable assignment of the edge constraint using source and target values from the ground facts of the derivation step.

Contracting nodes

The goal of this transformation is to reduce the number of nodes in the graph and extend the applicability of engines that operate on transition systems. This transformation is currently limited to nodes that do not participate in hyperedges and do not have a self-looping edge. Contraction means that a node v and all its edges are removed from the graph, and a new edge is added to the graph for every pair of edges (s, v) and (v, t). That is, every source of some v's incoming edge is now connected to every target of some v's outgoing edge. The label of the new edge is a conjunction of the labels of the two edges it replaces, with appropriate renaming of the variables. In the terminology of the underlying Horn clauses represented by the graph, this corresponds to the exhaustive application of the resolution rule on clauses where the node's predicate appears in the head and clauses where it appears in the body. Afterwards, all clauses containing the predicate are removed from the system. Note that while this transformation reduces the number of nodes in the graph, it replaces n incoming and m outgoing edges with $n \times m$ new edges. To avoid blowup in the number of edges, contraction can be limited to the case when either n or m is 1.

Summarization of a simple chain, described earlier, is a more efficient variant of gradual contraction of every intermediate node on a simple chain. Similarly to the summarization of simple chains, interpolation can be used for the backtranslation of a model when node contraction has been applied. Let v be the contracted node in the original graph. Given a model $\mathcal{M}$ for the transformed graph, we denote

$$A = \bigvee_{e \in in(v)} \mathcal{M}(src(e)) \wedge \mathcal{L}(e) \quad \text{and} \quad B = \bigvee_{e \in out(v)} \mathcal{L}(e) \wedge \neg\mathcal{M}(tgt(e)).$$

Since $\mathcal{M}$ is a model for the transformed graph, it follows that $A \wedge B$ is unsatisfiable. Extending the model $\mathcal{M}$ with an interpolant $I = \mathit{Itp}(A, B)$ as the interpretation for v's predicate yields a valid model for the original graph.

The backtranslation of a proof consists of replacing the derivation step corresponding to one of the new edges with two derivation steps corresponding to the original incoming-outgoing pair of edges. The missing ground instance of the predicate of the removed node is obtained in the same way as in the summarization of simple chains.

5.5 Back-end Engines of Golem

The core components of Golem that solve the problem of satisfiability of a CHC system are referred to as *back-end engines*, or just engines. The engines of Golem implement various algorithms from model checking and software verification that treat the problem of solving a CHC system as a *reachability* problem in the graph representation. There are currently five engines in Golem:

- BMC - Bounded Model Checking [29]
- KIND - k-induction [186]
- LAWI - Lazy Abstraction with Interpolants/Impact [157]
- Spacer [137]

- TPA/split-TPA - Transition Power Abstraction (Chapter 4)

Below, we describe each of the engines separately.

5.5.1 Bounded Model Checking

BMC engine in Golem works only over graphs that represent *transition systems.* The engine reconstructs the transition system's initial states, transition relation, and error states from the graph representation and then applies the basic bounded model checking algorithm [29]. It checks the existence of a path of gradually increasing lengths between the initial and error states. Formally, it checks the satisfiability of the formula

$$\mathit{Init}(x^{(0)}) \wedge \bigwedge_{i=0}^{n-1} \mathit{Tr}(x^{(i)}, x^{(i+1)}) \wedge \mathit{Bad}(x^{(n)})$$

for increasing value of n. If this formula is satisfiable for some value of n, a counterexample path of length n has been found. To speed up the satisfiability checks, it uses incremental SMT solving capabilities of OpenSMT.

By definition, BMC is capable of detecting the presence of a counterexample path (unsatisfiability of the corresponding CHC system), but not its absence.[6] The unsatisfiability proof of the CHC system is easily reconstructed from the model of the BMC formula, which gives the values of the variables at each step along the path.

5.5.2 k-induction

The KIND engine in Golem implements k-induction [186], a widespread technique in hardware and software verification. It was the first technique to extend the idea from BMC—the possibility to check the existence of a counterexample path with a satisfiability solver—to also *prove* the *absence* of counterexample paths. The basic idea of k-induction has been later refined [73] and adopted for software verification [21, 22, 43, 79].

The implementation in Golem follows the basic version of the algorithm from [186] and, as such, requires the input in the form of a transition system. The computation of the unsatisfiability witness proceeds the same way as in the BMC engine. On the other hand, the computation of the satisfiability witness is more complicated. This algorithm, by definition, computes k-inductive invariant of the transition system. However, to honor the semantics of the CHC system, 1-inductive invariant is required. Golem implements a procedure that first computes quantified 1-inductive invariant from the k-inductive one and then applies quantifier elimination (which is feasible for linear arithmetic) to obtain quantifier-free inductive invariant.

[6]Unless the initial states or the bad states are empty sets, i.e., already the formula *Init* or *Bad* is unsatisfiable.

5.5.3 Lazy Abstractions with Interpolants

Lazy Abstractions with Interpolants (LAWI) is an algorithm introduced by McMillan for verification of software [157]. It is sometimes known as the Impact algorithm, as that was the first tool where the algorithm was implemented. Later, it was also implemented in software verification tools Wolverine [143] and CPAchecker [22, 28].

The LAWI engine in Golem is a re-implementation of the original algorithm, as described in [157]. The original LAWI algorithm was designed for sequential programs. It operates on a program representation that maps straightforwardly to the graph representation of *linear* CHC systems in Golem. It analyzes possible paths through the graph, looking for a feasible path from *true* to *false*. It uses interpolation to learn from infeasible paths and compute invariants for individual nodes in the graph. The implementation in Golem leverages OpenSMT's ability to compute *path interpolants* from a single refutation proof of infeasibility for fast interpolant computation. As a faithful implementation of the original algorithm, the LAWI engine in Golem works on any graph without hyperedges, i.e., any linear, but not nonlinear, CHC systems.

5.5.4 Spacer

Spacer engine in Golem implements the algorithm originally named RecMC which was introduced in [136, 137] and implemented in the tool Spacer. Original Spacer is now the default fixed-point engine and Horn solver in Z3 [72]. Spacer in Z3, which we will refer to as Z3-Spacer, has been extended several times with various optimizations and support for theories beyond arithmetic since the original publication [135, 190, 191, 192].

The implementation in Golem follows the description from the journal publication [137]. Spacer algorithm heavily relies on efficient approximations for quantifier elimination. Typically, Craig interpolation is used to *over-approximate* quantifier elimination and *model-based projection* (MBP) [137] is used to *under-approximate* it. Golem relies on OpenSMT for interpolation but implements its own MBP procedure for integer and real linear arithmetic, based on the description from [33].

The advantage of the Spacer algorithm is that it works over any CHC system, even nonlinear ones. Nonlinear CHC systems can model programs with summaries, and in this setting, Spacer computes both under-approximating and over-approximating summaries of the procedures to achieve modular analysis of programs. Spacer is currently the only engine in Golem capable of solving nonlinear CHC systems.

5.5.5 Transition Power Abstraction

The TPA engine in Golem implements the algorithms introduced in Chapter 4. It can work in two modes: The first implements the basic TPA algorithm; the second implements the split-TPA algorithm. The implementation in Golem follows the algorithms faithfully. However, by default, the implementation uses model-based projection, not full quantifier elimination. Additionally, it uses incremental capabilities of OpenSMT to speed up the satisfiability queries. Consider the queries posed by Algorithm 4.2 on line 2. During its run, the algorithm makes a lot of these queries. For a single level n, the source and target states change, but the transition part $ATr^{\leq n}(X, X') \wedge ATr^{\leq n}(X', X'')$ is only ever refined, i.e., new conjuncts are added, but existing ones are not removed. Thus, these queries can be incremental when a separate solver is used for each level. Using incremental solving in this context significantly improved the algorithm's performance. However, as the internal implementation of incrementality in OpenSMT uses frame literals to track pushed and popped frames, these eventually clutter the solver. Thus, these solvers are torn down and rebuilt after a fixed number of incremental queries.

TPA engine can find counterexample paths for transition systems, which easily translate to unsatisfiability proofs for the corresponding CHC systems. For safe transition systems, it can discover safe k-inductive transition invariants. If a model for the corresponding CHC system is required, the engine first computes a quantified inductive invariant and then applies quantifier elimination to produce a quantifier-free inductive invariant.

The algorithms, as described in Chapter 4, work on transition systems. However, the engine in Golem already supports a more general class of graphs. Namely, it can analyze *chains of transition systems*. The graph representing a chain of transition systems is a sequence of nodes $\mathit{true}, v_0, v_1, \ldots, v_n, \mathit{false}$ where each node v_i has a self-loop and is connected to its predecessor and successor with a single edge. In the software domain, this represents a sequence of consecutive loops in a program. The idea is that each node represents a transition system and maintains its own TPA sequence for that system. Then, information flows along the chain. In the current implementation, nodes propagate reachable states forward (from *true* to *false*) and safe states backward. The direction of the flow could also be reversed, corresponding to the direction of information flow in the Spacer algorithm. In such a scenario, transition systems on the chain are asked various reachability queries for different initial and error states. However, their transition relations always remain the same. Thus, focusing on transitions rather than states is an advantage of the TPA algorithms. They learn information about transitions in the underlying system that is not invalidated when the initial or error states change. This information can be re-used across multiple reachability queries for the same transition relation.

5.6 OpenSMT for Golem: Integration and Improvements

Constrained Horn clauses, by design, require manipulation of *logical* terms and all approaches to CHC solving use an SMT solver as a sub-procedure in some way. Golem relies on OpenSMT [122] for term representation, term manipulation, SMT solving and interpolation. As part of the research on this thesis and the development of Golem, several contributions to OpenSMT were made to streamline the integration with Golem and improve the performance. The following are the changes most relevant for Golem.

The theory solver for linear arithmetic (LA) follows the simplex-based algorithm described in [84]. To support integer reasoning, OpenSMT initially implemented only the basic branch-and-bound algorithm [181], which introduces new branches only on variables from the original problem. Now, the LA solver also implements a basic version of the cuts-from-proofs algorithm [78] that computes branches on general terms. This addition significantly improved OpenSMT's performance on integer problems and substantially decreased the number of benchmarks where OpenSMT exhibits diverging behaviour.

The optimizations, which contributed the most to the performance improvement of OpenSMT's LA solver, are theory suggestions for deciding the polarity of a theory atom in the SAT solver and quasi-basic variables in the tableau implementation.

- *Theory suggestions*: During the search for a satisfying assignment, a SAT solver makes *decisions*: picks an unassigned atom and sets it either to true or false. When choosing the atom's *polarity* (whether to assign it to true or false), OpenSMT originally defaulted to positive polarity. Now, for the atoms of the theory of linear arithmetic, it picks the polarity that is consistent with the current assignment maintained by Simplex. The intuition is that choosing a polarity which yields a constraint that is not consistent with the current assignment in Simplex requires unnecessary work: an update to the assignment and potentially many pivoting operations over the tableau to make the assignment consistent again. Choosing a consistent assignment avoids unnecessary work and leads to significant performance improvement.

- *Quasi-basic variables*: In the typical implementation of the incremental Simplex algorithm in SMT solvers, all linear expressions from the original problem participate in the creation of the tableau. As constraints are asserted and retracted during the SAT solver search, bounds on the linear expressions are activated and deactivated. New bounds may violate the current satisfying assignment maintained by Simplex, leading to an update of variables' values and pivoting operations over the tableau. Originally, OpenSMT was updating all affected rows during a pivot operation. However, if no bound has been asserted yet for a row, it cannot participate in a conflict, and its update can be delayed. Moreover, if backtracking is triggered before the execution of the update, the update can be avoided altogether. This idea to delay certain updates in the tableau has already been implemented in Z3, which referred to the basic variables without an asserted bound as *quasi-basic* variables. While in Z3, to the best of our knowledge, quasi-basic variables are not used by default, in OpenSMT, they led to a significant performance improvement of the Simplex algorithm.

Two architectural changes were necessary for streamlined integration and efficient use of OpenSMT in the infrastructure of Golem. First, a clear separation between the term manager and core solver has been made. The separation allowed multiple solver instances to share a single term manager. Golem uses a single instance of the term manager across the whole architecture: for representation of the CHC system, for term rewriting in the preprocessing, and for all solver instances in the back-end engine. TPA algorithms especially take advantage of this feature as they maintain multiple solvers at the same time.

The second change was improved interpolation support, especially making interpolation a runtime option instead of a compile-time option. Moreover, the interpolation module of OpenSMT was extended to work in the context of incremental SMT solving. Previously, OpenSMT had support for incremental solving and interpolation but not the combination of both features. Interpolation for incremental SMT solving is used mainly in the TPA algorithms, which rely heavily on interpolation. However, it can utilize incremental solving to maximize efficiency and avoid repeating redundant work.

Overall, OpenSMT was the key to quickly developing an efficient Horn solver. Relying on the existing data structures of OpenSMT for term representation facilitated the work on the solving algorithms. The term rewriting capabilities, decision procedures and interpolation algorithms available in OpenSMT enabled rapid prototyping of the back-end engines. In the other direction, Golem stresses various parts of OpenSMT, e.g., efficient term rewriting, solving both a small number of complex queries and a large number of easy queries, efficient incremental solving and interpolation. Heavy use of OpenSMT uncovered several performance bottlenecks in the solver. Uncovering and fixing these issues improved OpenSMT not only for Golem but also for other applications.

5.7 Experiments

This section presents several experiments related to Golem and its engines. First, we discuss the experiments and results related to the implementation of the TPA and split-TPA algorithms from Chapter 4 in the TPA engine of Golem. Then we compare individual engines of Golem among themselves and with the state-of-the-art tools. All experiments were conducted on a machine with an AMD EPYC 7452 32-core processor and 8x32 GiB of memory.

5.7.1 Evaluation of the TPA algorithms

The first set of experiments uses only the TPA algorithm and focuses on detecting deep counterexamples. The second set additionally evaluates the split-TPA algorithm and its ability to prove safety of transition systems. Experiment setup and the presentation follow the respective publications on TPA and split-TPA [35, 36].

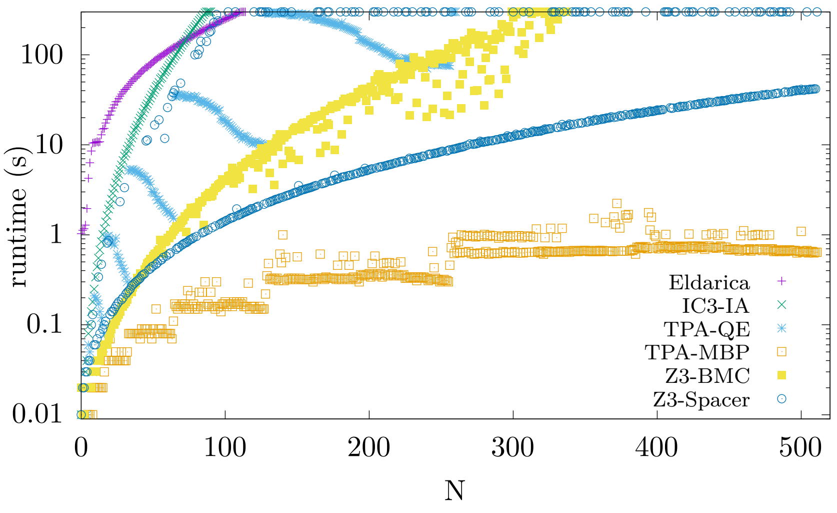


Figure 5.3. Runtime for motivating example for N from 1 to 511 (log y-axis)

Detecting Deep Counterexamples with TPA

In the first set of experiments, we evaluate the ability of TPA to detect long counterexamples and compare its performance with state-of-the-art tools Eldarica 2.0.6 [117], IC3-IA 20.04.1 [59] and Z3 4.8.12 [72] (using both its BMC [29] and Spacer [137] engines). These were the top competitors in CHC-COMP 2020 and 2021 [95, 174]. We use two versions of the TPA algorithm in the experiments. One uses full quantifier elimination as is denoted as TPA-QE. The other under-approximates quantifier elimination with model-based projection and is denoted as TPA-MBP. All benchmarks are encoded as CHC satisfiability problems using the format of CHC-COMP. Since IC3-IA's input format differs, all CHC benchmarks were translated to VMT format using the automated tool packaged with IC3-IA.[7]

First, we investigated the scalability of the algorithms with respect to the length of the counterexample. For this purpose, we used the parametrized transition system from our motivating example in Section 4.2. The counterexample in this system has length $2N$, and we ran the tools on instances for N ranging from 1 to 511. The timeout was set to 300 seconds. Figure 5.3 shows the runtime of the tools for the given value of N.

TPA-MBP was able to report all instances as unsafe, requiring *less than two seconds* for each instance. Eldarica, IC3-IA and Z3-BMC exhibit relatively stable pattern where the performance decreases rapidly with increasing N. Z3-Spacer, on the other hand, exhibits a curious behaviour where it can solve most of the instances (even though it is slower than TPA-MBP by at least an order of magnitude), but on a relatively large number of instances, it times out.[8] Finally, TPA-QE performed significantly worse than

[7]Artifact for these experiments is available at `https://doi.org/10.5281/zenodo.5815911`

[8]The authors of Z3-Spacer confirmed this behaviour. It seems the root cause is an interpolant which

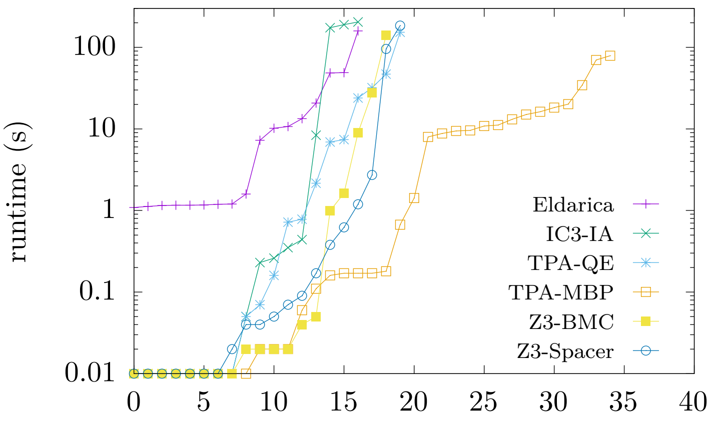


Figure 5.4. Results on 54 multi-phase unsafe benchmarks

TPA-MBP, which benefited from the fact that the reason why shorter counterexamples do not exist can be summarized relatively easily.

Continuing this line of experiments, we considered a second set of benchmarks representing instances of challenging safety properties of *multi-phase* loops [185], which are known to be difficult to analyze by state-of-the-art techniques. We took 54 safe multi-phase benchmarks from CHC-COMP repository[9] and then, for each benchmark, created its unsafe version with a minor modification of the safety property.[10] In most cases, this was done by negating one of the conjuncts of the property. In a few cases, this resulted in a simple benchmark with a very short CEX (< 10 steps), but in most cases, the minimal counterexample is much larger, ranging from hundreds to tens of thousands of steps. There are even a few extremes where the minimal counterexample requires hundreds of thousands or even millions of steps. With the timeout of 300 seconds, out of 54 benchmarks, TPA-QE solved 20 and TPA-MBP solved 35 benchmarks, beating the other tools, among which Z3-SPACER performed the best, solving 20 benchmarks. The results are summarized in Figure 5.4, where the number of solved benchmarks by each tool is plotted against the time needed for their solving.

Overall, TPA-MBP solved 15 benchmarks that none of the other tools was able to solve and, in general, could be one or two orders of magnitude faster. There were two noticeable exceptions: benchmark 24 was uniquely solved by Z3-SPACER, and IC3-IA uniquely solved benchmark 39. In the latter case, our tool suffered from incompleteness in the decision procedure of OpenSMT for integer arithmetic, while in the former case, the interpolation was not producing good abstractions, and TPA-MBP suffered from the need

can either help the SPACER algorithm converge quickly or hinder its progress significantly, depending on which of the possible interpolants is computed by Z3.

[9] `https://github.com/chc-comp/aeval-benchmarks`

[10] Benchmarks available at `https://github.com/blishko/chc-benchmarks`.

Benchmark suite	split-TPA	TPA	Z3Spacer	GSpacer	Eldarica
multi-phase safe	19 (7)	12 (0)	6 (0)	24 (3)	26 (4)
multi-phase unsafe	37 (3)	35 (2)	20 (0)	17 (0)	17 (0)

Table 5.1. Summary of the experiments on multi-phase benchmarks: Solved (unique) instances out of 54 benchmarks.

for frequent refinements. We also examined the solved benchmarks for the length of the minimal counterexample they admit. The results are in line with the observations from our first experiments: Other tools could only solve benchmarks with a counterexample of up to a thousand steps (1001 steps in benchmark 17 solved by Z3-Spacer). TPA-QE matched this performance (1001 steps in benchmark 27), but TPA-MBP managed to solve benchmarks with a counterexample of more than *ten thousand* steps (17650 in benchmark 42). Thus, our technique significantly improves upon state-of-the-art with respect to the length of the counterexample it can detect.

Proving Safety with split-TPA

In the next set of experiments, we focused on evaluating the implementation of split-TPA for its ability to prove systems safe. However, as split-TPA implements a more complex procedure for answering bounded reachability queries, we also ran experiments on the unsafe version of the problems. Note that compared to the previous set of experiments, these are more recent and, as such, use later versions of the tools. In particular, here we compared Golem 0.1.0 using OpenSMT 2.3.2, Eldarica 2.0.8 [117], Z3-Spacer [137] implemented in Z3 4.8.17 [72], and GSpacer [190] a more recent version of Spacer enriched with global guidance.[11]

For the evaluation, we used the same set of benchmarks representing multi-phase loops as in the previous experiment, both safe and unsafe versions. The results are summarized in Table 5.1, and times for each tool/benchmark pair are given in Table 5.2.

Regarding safety, Table 5.1 shows that split-TPA overall solved seven more benchmarks than TPA, but still less than GSpacer or Eldarica. However, it solved seven benchmarks *uniquely* (the other competitors did not solve them). This indicates that split-TPA is quite orthogonal to the existing techniques for proving safety.

The results on unsafe benchmarks show that split-TPA not only preserves the capability of TPA to detect deep counterexample, but it was even able to outperform it by solving two more benchmarks overall.

[11]Artifact for these experiments is available at `https://doi.org/10.5281/zenodo.6988735`

Ben.	SPLIT-TPA	TPA	Z3SPACER	GSPACER	ELDARICA
01	**26.28**	TO	TO	TO	TO
02	TO	TO	133.28	**<1**	TO
03	TO	TO	TO	TO	**1.33**
04	TO	TO	TO	**<1**	3.70
05	**<1**	**<1**	**<1**	**<1**	1.19
06	TO	TO	TO	TO	**3.95**
07	TO	TO	TO	**<1**	1.32
08	TO	TO	TO	TO	TO
09	TO	TO	TO	TO	TO
10	TO	TO	TO	TO	TO
11	TO	TO	TO	**5.68**	TO
12	TO	TO	TO	TO	**1.62**
13	**<1**	**<1**	ERR	**<1**	1.16
14	**53.94**	TO	TO	TO	118.78
15	TO	TO	TO	TO	TO
16	TO	TO	TO	TO	TO
17	**<1**	37.50	TO	**<1**	7.53
18	**<1**	**<1**	TO	**<1**	3.66
19	TO	TO	**<1**	**<1**	1.22
20	TO	TO	TO	TO	TO
21	**<1**	10.39	TO	**<1**	15.45
22	TO	TO	TO	TO	TO
23	**<1**	**<1**	ERR	**<1**	1.79
24	TO	TO	TO	TO	TO
25	TO	45.93	TO	TO	**9.33**
26	2.60	1.55	TO	**<1**	TO
27	TO	TO	TO	TO	TO
28	**<1**	TO	TO	TO	1.61
29	**3.94**	TO	TO	118.98	34.22
30	TO	TO	TO	TO	**20.48**
31	TO	TO	TO	**<1**	1.60
32	TO	TO	TO	**11.49**	TO
33	TO	TO	TO	TO	TO
34	TO	TO	TO	**<1**	5.86
35	TO	TO	TO	**<1**	1.80
36	**<1**	**<1**	TO	**<1**	1.92
37	**<1**	**<1**	**<1**	**<1**	14.33
38	TO	**<1**	TO	**<1**	1.36
39	TO	TO	67.41	58.73	**2.48**
40	**109.05**	TO	TO	TO	ERR
41	TO	TO	TO	TO	TO
42	TO	TO	TO	**<1**	4.37
43	TO	TO	TO	**5.20**	TO
44	TO	TO	TO	TO	TO
45	TO	TO	TO	TO	TO
46	TO	288.20	13.07	**<1**	1.28
47	TO	TO	TO	TO	TO
48	**47.00**	TO	TO	TO	TO
49	**122.96**	TO	TO	TO	TO
50	TO	TO	TO	TO	TO
51	TO	TO	TO	TO	TO
52	**235.24**	TO	TO	TO	TO
53	**147.28**	TO	TO	TO	TO
54	**133.63**	TO	TO	TO	TO

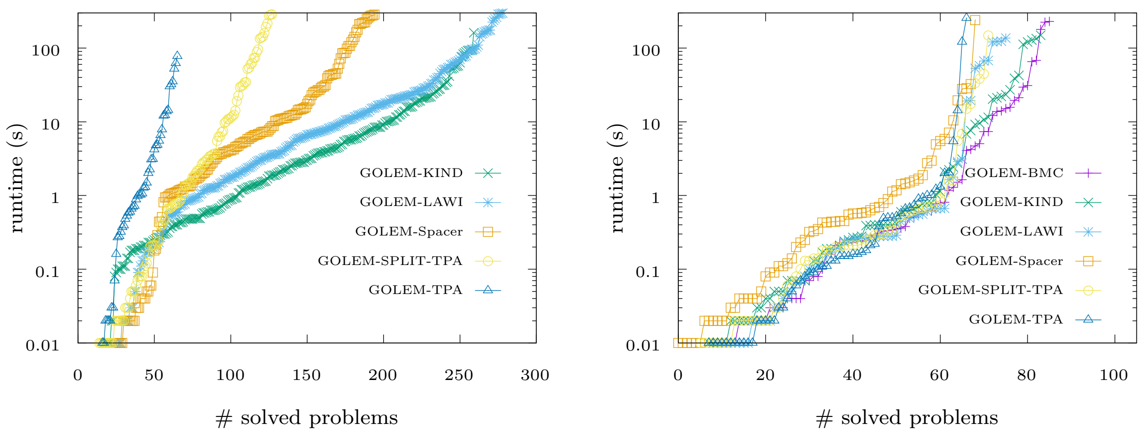
Ben.	SPLIT-TPA	TPA	Z3SPACER	GSPACER	ELDARICA
01	14.53	**10.12**	TO	TO	TO
02	**<1**	**<1**	1.25	TO	TO
03	**<1**	**<1**	**<1**	**<1**	1.16
04	TO	TO	TO	TO	TO
05	**<1**	**<1**	**<1**	**<1**	1.18
06	TO	TO	TO	TO	TO
07	TO	TO	TO	TO	TO
08	TO	TO	TO	TO	TO
09	TO	TO	TO	TO	TO
10	**20.40**	233.78	TO	TO	TO
11	**152.28**	TO	TO	TO	TO
12	TO	TO	TO	TO	TO
13	**<1**	**<1**	**<1**	**<1**	1.13
14	**<1**	**<1**	**<1**	8.91	89.78
15	TO	TO	TO	TO	TO
16	TO	TO	TO	TO	TO
17	**14.84**	15.81	181.59	TO	TO
18	**<1**	**<1**	**<1**	**<1**	1.57
19	**<1**	**<1**	**<1**	**<1**	20.74
20	TO	TO	TO	TO	TO
21	**<1**	**<1**	**<1**	**<1**	10.63
22	TO	TO	TO	TO	TO
23	**<1**	**<1**	**<1**	**<1**	1.17
24	**<1**	TO	96.64	TO	TO
25	**<1**	**<1**	**<1**	**<1**	1.19
26	2.01	**1.46**	TO	TO	TO
27	**<1**	**<1**	TO	TO	TO
28	**<1**	**<1**	TO	TO	162.43
29	**<1**	**<1**	2.76	32.56	45.75
30	**<1**	**<1**	**<1**	**<1**	10.22
31	TO	TO	TO	TO	TO
32	**<1**	**<1**	**<1**	**<1**	7.17
33	**<1**	**<1**	**<1**	**<1**	1.21
34	**<1**	**<1**	**<1**	**<1**	1.15
35	**<1**	**<1**	**<1**	**<1**	1.20
36	16.68	**14.45**	TO	TO	TO
37	**<1**	**<1**	**<1**	**<1**	13.37
38	**262.18**	TO	TO	TO	TO
39	TO	TO	TO	ERR	TO
40	**<1**	**<1**	**<1**	133.07	ERR
41	TO	**4.60**	TO	TO	TO
42	**18.31**	40.39	TO	TO	TO
43	TO	TO	TO	TO	TO
44	**34.18**	TO	TO	TO	TO
45	TO	TO	TO	TO	TO
46	TO	**239.05**	TO	TO	TO
47	**5.71**	6.79	TO	TO	TO
48	17.52	**12.10**	TO	TO	TO
49	32.59	**12.49**	TO	TO	TO
50	TO	TO	TO	TO	TO
51	**6.71**	11.57	TO	TO	TO
52	**70.83**	82.43	TO	TO	TO
53	57.42	**33.00**	TO	TO	TO
54	40.74	**15.15**	TO	TO	TO

Table 5.2. Full results on safe (left) and unsafe benchmarks (right). TO: timeout; ERR: memory out or other inconclusive answer.

5.7.2 Evaluation on CHC-COMP Benchmarks

The benchmark collections of CHC-COMP represent a rich source of problems from various domains.[12] We evaluate the performance of GOLEM and its engines on different categories of these benchmarks. For these experiments, GOLEM 0.3.1 was used, which is a more recent version than in the experiments of the previous section. The timeout for all experiments was set to 300 seconds.

[12] https://github.com/orgs/chc-comp/repositories



(a) SAT problems *(b)* UNSAT problems

Figure 5.5. Performance of Golem on the 498 benchmarks of the LRA-TS track

	BMC	KIND	LAWI	Spacer	split-TPA	TPA
SAT	0	260	279	190	127	66
UNSAT	86	84	76	69	72	67

Table 5.3. Number of solved benchmarks from LRA-TS track for each engine of Golem

Category LRA-TS

The first category of benchmarks considered is the LRA-TS category of CHC-COMP. It consists of problems that model the safety of transition systems using linear real arithmetic as the background theory. There are 498 unique benchmarks in this category, and all were used in the 2021 and 2022 editions of CHC-COMP. All engines of Golem can solve benchmarks in this category. The results for this benchmark set are presented in Figure 5.5 and Table 5.3.

Figure 5.5 plots, for each engine, the number of solved benchmarks (x-axis) within the given time limit (y-axis, log scale). The results are split into satisfiable and unsatisfiable problems. As the BMC engine does not make any attempt beyond trivial cases to solve satisfiable instances, it did not solve any satisfiable benchmark in this experiment and is omitted from the corresponding plot. Table 5.3 summarizes the results with the total number of benchmarks solved for each engine.

For the unsatisfiable problems, the performance is similar across all engines, with the BMC engine performing the best. This is expected as the BMC algorithm does not make any extra effort to discover invariants and focuses purely on the search for a counterexample trace in the transition system. However, the extra check for k-inductive invariants on top of the BMC-style search for counterexamples, as implemented in the

Golem-TPA	Golem-split-TPA	Golem-LAWI	Golem-Spacer	Z3-Spacer	Eldarica
31	44	12	18	16	36

Table 5.4. Number of solved benchmarks from `extra-small-lia` subcategory

KIND engine, incurs a relatively small overhead in unsatisfiable problems compared to the impressive success in solving satisfiable problems.

For the satisfiable problems, the performance varies widely across the different engines. With the timeout of 300 seconds, the LAWI engine solved the largest number of satisfiable problems, followed closely by the KIND engine. The Spacer engine solved significantly fewer benchmarks than LAWI and KIND but still outperformed both algorithms of the TPA engine.

We make the following observations regarding the performance of the two TPA-based algorithms. Firstly, the results show the superiority of split-TPA over the basic TPA algorithm for satisfiable problems, with comparable performance on unsatisfiable problems. This is in line with the results on problems representing multi-phase loops. Secondly, the performance of TPA-based algorithms lacks behind the other engines. The nature of the benchmarks in the LRA-TS category partially explains this underperformance. These benchmarks come from relatively few sources (see [95]) and often contain many boolean variables and a transition relation with complex boolean structure. These factors complicate the discovery of useful summaries for multiple transition steps in the TPA-based algorithms. However, as we will see next, there is a benchmark set in LIA-Lin category where TPA-based algorithms excel.

Category LIA-Lin

The next category of benchmarks is the LIA-Lin category of CHC-COMP. These are linear systems of CHCs with linear integer arithmetic as the background theory. There are many benchmarks in this category, and for the evaluation at the competition, a subset of benchmarks is selected (see [71, 95] for the selection process). We evaluate the LAWI and Spacer engines of Golem (the engines capable of solving general linear CHC systems) on the benchmarks selected at CHC-COMP 2022 and compare their performance with Z3-Spacer (Z3 4.11.2) and Eldarica 2.0.8. However, we first examine a specific subcategory of LIA-lin, namely `extra-small-lia`[13]. The benchmarks in this subcategory are also solvable by Golem's TPA engine, with compelling results.

The benchmarks in `extra-small-lia` subcategory are syntactically relatively simple, and all are satisfiable. Semantically, they represent one or more loops that exercise various properties of linear integer arithmetic and often require invariants that are hard to find even by state-of-the-art algorithms. Overall there are 55 benchmarks in this subcategory, and the performance of the considered tools is summarized in Table 5.4.

[13]`https://github.com/chc-comp/extra-small-lia`

	Golem-LAWI	Golem-Spacer	Z3-Spacer	Eldarica
SAT	131	184	211	183
UNSAT	77	82	96	60

Table 5.5. Number of solved benchmarks from LIA-Lin track of CHC-COMP 2022.

	Golem-Spacer	Z3-Spacer	Eldarica
SAT	239 (4)	248 (13)	221 (6)
UNSAT	124 (2)	139 (5)	122 (0)

Table 5.6. Number of solved benchmarks from LIA-Nonlin track of CHC-COMP 2022. Number of *uniquely* solved benchmarks in parenthesis.

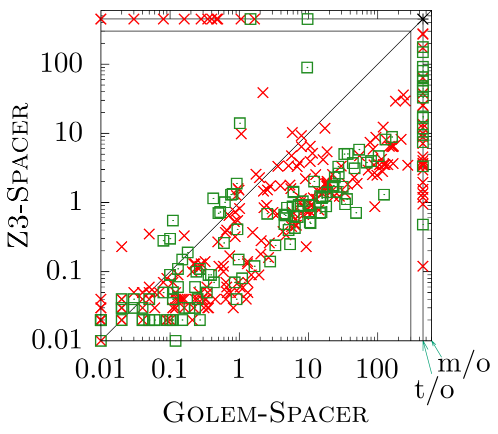
The TPA engine of Golem solves many more benchmarks than its LAWI and Spacer engine in this subcategory. The split-TPA variant also beats the competitors, solving 44 out of 55 benchmarks, while Eldarica, the second best, solves 36 of these benchmarks.

For the whole LIA-Lin category, 499 benchmarks were selected in the 2022 edition of CHC-COMP [71]. The performance of the LAWI and Spacer engines of Golem, Z3-Spacer and Eldarica on this selection is summarized in Table 5.5. In this category, the Spacer engine of Golem significantly outperforms the LAWI engine. Moreover, it also outperforms Eldarica due to the much better performance on unsatisfiable instances. However, there is still a significant performance gap compared to Z3-Spacer.

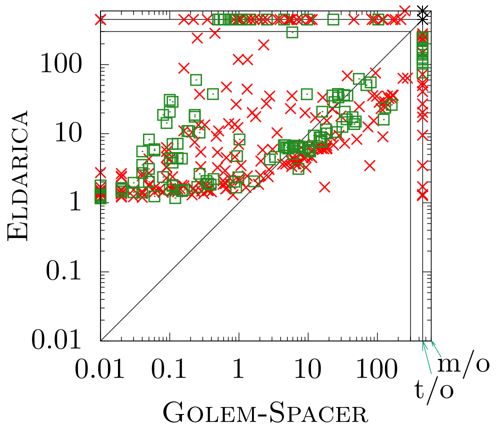
Category LIA-Nonlin

Finally, we consider the LIA-Nonlin category of benchmarks of CHC-COMP, which consists of *nonlinear* systems of CHCs with linear integer arithmetic as the background theory. Similarly to LIA-Lin, there is a large number of benchmarks in this category collected in various repositories of CHC-COMP. For the competition, a subset of the benchmarks is selected (see [71, 95] for the selection process). For the 2022 edition of CHC-COMP, 456 benchmarks were selected. Spacer is the only engine in Golem capable of solving nonlinear CHC systems; thus, we focus on a more detailed comparison of its performance against Z3-Spacer and Eldarica. The results of the experiments are summarized in Table 5.6 and Figure 5.6.

The summary results are in line with the results from CHC-COMP 2022 [71], even though we used a smaller timeout and ran the experiments in a different environment. Overall, Golem solved fewer problems than Z3-Spacer but more than Eldarica. A detailed comparison is depicted in Figure 5.6. For each benchmark, its data point on the plot reflects the runtime of Golem (x-axis) and the runtime of the competitor (y-axis). The plots suggest that the performance of Golem is often orthogonal to Eldarica,[14] but highly correlated with the performance of Z3-Spacer. This is not surprising as the



(a) Golem vs Z3-Spacer



(b) Golem vs Eldarica

Figure 5.6. Comparison of Golem against state-of-the-art tools on LIA-Nonlin track on SAT (×) and UNSAT (⊡) instances.

Spacer engine in Golem is built on the same core algorithm. Even though Golem is often an order of magnitude slower than Z3-Spacer, there is a non-trivial amount of benchmarks on which Z3-Spacer times out, but which Golem solves fairly quickly.

5.8 Future Work

There are multiple possible directions for Golem's future development. Some require mostly engineering effort, while others could lead to interesting new research topics.

At the front end, support for more input formats can be added. For example, Z3's Datalog input format represents systems of Horn clauses more compactly than the standard format used in CHC-COMP. A front end for VMT input format [62] would allow Golem to access new benchmark sets. It would also be the first step towards extending Golem solving capabilities beyond safety properties.

In the preprocessing phase, Golem could apply more transformation passes, such as cone-of-influence reduction and constant propagation. Moreover, some of the already implemented transformation passes do not support nonlinear clauses (hyperedges in the graph representation), which limits their applicability for nonlinear CHC systems. Improving this support would improve the performance of Golem on nonlinear problems.

In the back end, more engines could be added, and the existing ones could be improved. Natural candidates for new engines are pd-kind algorithm [129] and algorithms based on (implicit) predicate abstraction, such as those implemented in the tools Eldarica [117]

[14]The requirement of the Java Virtual Machine runtime environment adds approximately one second overhead to each run of Eldarica.

and IC3-IA [59]. Regarding the existing engines, the performance of Spacer engine could be improved by adding missing optimizations and global guidance [190]. The TPA engine could be extended to handle more general CHC systems. Investigating the combination of the TPA-style analysis of loops with Spacer-like search can yield an efficient algorithm for modular analysis of complex nonlinear CHC systems. Such an algorithm would work as an *unbounded* Spacer-like traversal of the hypergraph. While the traversal in Spacer is bounded, with gradually increasing bound, combination with TPA would allow it to handle self-loop nodes directly. Similarly, the LAWI engine could be extended to support nonlinear CHC systems, following the approach implemented in the tool Duality [159, 160].

Extending the support for SMT theories in Golem beyond linear arithmetic would open up more opportunities for further research in tailored decision and interpolation procedures. With support for arrays, Golem could be used for a much broader set of problems from software verification, while with support for bit-vectors, it could also be used for word-level hardware model checking. Golem can already be used as the back end of C verifier Korn [87]. With support for more complex SMT theories, it could also serve as the back end, for example, for SolCMC [5].

5.9 Related Work

There is a fair amount of Horn solvers, model checkers, and software verifiers from both academia and industry. We discuss only the ones closely related to Golem.

According to the results of the last couple of editions of CHC-COMP, the most prominent Horn solvers are Eldarica [117] and Spacer [137]. Golem follows the trend of tigher integration with the underlying SMT solver outlined by these two Horn solvers, but offers more choices for the back-end solving algorithm.

Eldarica has been developed since 2013. It implements many sophisticated transformation passes to simplify the input CHC system. The preprocessing phase in Eldarica is followed by its main reasoning engine, which combines Predicate Abstraction [100] with Counterexample-Guided Abstraction Refinement (CEGAR) [63] to solve the resulting system. The algorithm is described in detail in [176]. Eldarica relies on Craig interpolation [68] to compute predicates for predicate abstraction. Moreover, it controls the form of the interpolants with *interpolation abstraction* [146, 177]. Besides the standard input format used in CHC-COMP, Eldarica accepts Horn clause systems written in Prolog style. Additionally, it accepts as input Numerical Transition Systems [116], programs in a fragment of the C language and networks of timed-automata in a C-like language [118]. Eldarica uses Princess [173] for SMT solving and Craig interpolation. It supports multiple SMT theories, including integers, bit-vectors, arrays and algebraic data types. Eldarica is used as a back-end solver for several verification tools, e.g., TriCera [88], SolCMC [5], Korn [87], CoCoSim [65], JayHorn [132], Vac [96].

Spacer is likely the most-known Horn solver at the moment. The updates since its beginning [136, 137] include global guidance [191] and improved support for the theory of arrays [135], bit-vectors [190], algebraic data types and recursive functions [192]. It now lives as the default fixed-point engine in Z3 [72]. The terminology of "fixed point" comes from Datalog [1], which was the original front end supported by Z3 for this class of problems.[15] Like ELDARICA, Z3 implements several preprocessing passes over the system of Horn clauses. They simplify the input system before it is passed to SPACER. Unfortunately, not all transformations have a corresponding backtranslation, so some transformations need to be disabled to obtain full proof of unsatisfiability, for example, clause inlining. SPACER is used as the back-end solver for several verification tools, e.g., SEAHORN [107], SolCMC [5], KORN [87], RUSTHORN [154], COCOSIM [65], SOLTYPE [187]. Recently, it was also used in a new reduction approach for reasoning about data structures with CHCs [89].

Besides ELDARICA and SPACER, there exist several other Horn solvers: HOICE [52] implements a machine-learning-based technique for solving CHC systems, adapted from the ICE framework originally developed for discovering inductive invariants of transition systems [51]. FREQHORN [92, 93] implements an algorithm based on Syntax-Guided Synthesis (SyGuS) [9] to discover the interpretations of the unknown predicates. It combines the basic SyGuS approach with data derived from unrollings of the CHC system. ULTIMATE TREEAUTOMIZER [75] implements automata-based approaches to CHC solving [130, 195]. PCSAT [189] is a solver for a general class of second-order constraints on predicate and function variables. CHC satisfiability problem is just a subset of this general class.

Many model checkers and software verifiers implement similar algorithms as Horn solvers in general and as GOLEM in particular. While model checkers typically verify safety properties of transition systems, there are approaches for dealing with more complex CHC systems. Linear systems of Horn clauses can be encoded as transition systems [47]; this approach has been successfully applied for the model checker IC3-IA [59, 69]. An approach to solve nonlinear CHC systems using a solver for linear CHCs has also been proposed [131], but it does not appear to be competitive so far. Model checker PONO [149] is a highly configurable and extensible tool that implements several model-checking algorithms as the back-end reasoning engines, similar to GOLEM. Besides the standard algorithms such as BMC [29] and k-induction [186], it also offers an implementation of McMillan's interpolation-based model checking [155] and an impressive list of IC3-based algorithms. Although all its engines operate over transition systems, PONO could be used to solve more general CHC systems with the help of the reduction techniques mentioned above. Compared to PONO, GOLEM can solve nonlinear CHC systems directly. This is crucial for domains such as software verification where programs with functions are naturally modeled with nonlinear CHC systems.

In software verification, software verifiers implement many algorithms that originated in the context of model checking. Moreover, almost all verifiers now implement more than

[15] Datalog programs are written as a set of Horn clauses.

one technique, as described in the 2022 report from Software Verification Competition (SV-COMP) [19]. As an example, CPAchecker [25] uses a framework of Configurable Program Analysis (CPA) [23] to implement a group of algorithms similar to Golem engines: BMC, k-induction, Impact, McMillan's original interpolation-based model checking algorithm, and IC3/PDR [20, 21, 22, 26, 28]. However, traditional software verifiers use SMT solvers as black boxes. The tight coupling of Golem with its underlying SMT solver provides full control over their interaction. It allows Golem to detect and resolve (or at least avoid) many of the pitfalls and bottlenecks in the SMT solver.

5.10 Conclusion

In this chapter, we presented Golem, an efficient Horn solver with multiple back-end engines and tight integration with the underlying SMT solver OpenSMT. In our work, Golem was instrumental in prototyping the TPA algorithms described in Chapter 4; however, there is much more potential for further applications.

The common framework makes it easy to compare different algorithms for model checking and CHC solving directly. It eliminates unrelated differences when comparing algorithms in different tools, for example, a different SMT solver, which plays a significant role in the efficiency of these algorithms. Moreover, using the existing infrastructure of Golem, new back-end engines can be added easily. This possibility facilitates the reproduction of results for novel algorithms, which increases trust in the original results and validates the contributions of such novel algorithms.

There are a lot of possible applications of Golem than can build on the current foundations. We have outlined several directions in Section 5.8. In the short term, we plan to improve and extend Golem to serve as the back end for software verifiers, such as Korn [87] and SolCMC [5]. In the longer term, Golem can further improve the tool support for CHC solving and help academia and industry deal with complex tasks from verification and other domains by leveraging the powerful framework of constrained Horn clauses. Additionally, it can serve as a research tool for experimenting with novel techniques in SMT solving and interpolation. One such project, which evaluates the benefits of lookahead-based SMT solving for interpolation and model checking, is currently underway.

Chapter 6

Cooperative Parallelization Approach for Property-directed k-induction

A sequential approach to model checking has inherent limitations due to the undecidable nature of the problem. Different model-checking algorithms are better suited for different types of problems. Moreover, even within a single algorithm, many parameters and strategies can be adjusted. Such tuning is known to affect dramatically not only the algorithm run time but also its convergence. The variety of algorithms and heuristics naturally leads to the *parallelization* of model-checking algorithms. Recent results [50, 150, 162] show that even a simple algorithm portfolio leads to substantial improvements in performance. However, the key to a truly scalable solution is the sharing of information among the solvers of the portfolio (see, e.g., [150]), a usually much more complicated task than constructing a portfolio.

This chapter describes an abstract framework IcE/FiRE that generalizes the concepts from a recently introduced class of model-checking algorithms that combine the strength of k-induction with IC3-style search for safe inductive invariants [106, 129]. The framework consists of two components, the *induction-checking engine* and the *finite reachability engine*. We show that the components are general enough to enable not only *internal* learning but also *external* learning. While internal learning happens as part of the execution of the sequential algorithm, external learning happens in the parallel setting where multiple instances *share* information discovered about the system under analysis. We describe how to arrive at a parallel version of an efficient model checking algorithm PD-KIND, implement it in SMTS framework for parallel solving [152], and show with a robust experimental analysis that PD-KIND and related algorithms can profit significantly from this type of parallelization already in a multi-core environment.

Internally, PD-KIND relies on interpolation for learning facts about reachability in the analyzed system; thus, the results of Chapter 3 provide a possibility to compute different interpolants and consequently learn different facts about the system. As part of the experiments, we evaluate the benefit of applying decomposed Farkas interpolants in this setting. The results show that computing diverse reachability facts by means of different interpolation algorithms significantly contributes to the performance improvement of parallel PD-KIND.

6.1 Preliminaries

In this chapter we use the terminology related to transition systems as given in Section 2.5. Given a transition system $\mathcal{S} = \langle I, T \rangle$, a state formula P and a set of state formulas $\mathcal{F}$, we say that P is $\mathcal{F}^k$*-inductive* if

$$\bigwedge_{i=0}^{k-1} ((\mathcal{F}(X_i) \wedge P(X_i)) \wedge T(X_i, X_{i+1})) \implies P(X_k) \tag{6.1}$$

If $\mathcal{F} = \{P\}$ and P is a $(k-1)$-invariant, then P is a *k-inductive invariant* of $\mathcal{S}$, meaning it is valid in all reachable states of $\mathcal{S}$. When P is not $\mathcal{F}^k$-inductive, the negation of (6.1) is satisfiable and each satisfying assignment defines a trace $\langle s_0, \ldots, s_k \rangle$ of k+1 states called a *counterexample to (k-)induction* (CTI). We say that a CTI is reachable in $\mathcal{S}$ when s_0 is reachable. A central task of the algorithm presented in this paper is to check if elements of $\mathcal{F}$ are $\mathcal{F}^k$-inductive. Checking this for an element P of $\mathcal{F}$ and placing P to another set $\mathcal{G}$ if P is $\mathcal{F}^k$-inductive is referred to as *pushing* P to $\mathcal{G}$.

6.2 The IcE/FiRE Framework

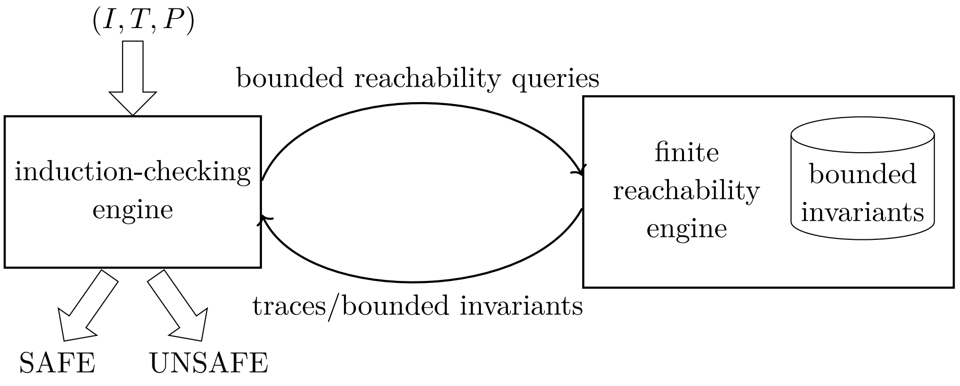

(I, T, P)
bounded reachability queries
induction-checking engine
finite reachability engine
bounded invariants
traces/bounded invariants
SAFE
UNSAFE

Figure 6.1. The IcE/FiRE framework for solving safety of transition systems

This section formalizes a general approach for checking the safety of symbolically represented transition systems in a way that allows us to present our parallelization techniques naturally. The approach splits the reasoning about safety into two separate components (Figure 6.1). The main component is an *induction-checking engine* (IcE), also referred to shortly as an induction engine. The goal of the induction engine is to decide the safety problem. It searches for a k-inductive strengthening of the property

P being checked. If it finds such a strengthening, it reports the system as safe. During the search, it may discover that no such strengthening exists since the negation of the property is reachable from the initial states. In this case, it reports the system as unsafe. To make progress in its search, to remove spurious counterexamples to induction, and to confirm real ones, IcE relies on the services of the second component, *finite reachability engine* (FiRE). The role of FiRE is to answer *bounded reachability queries* issued by IcE. Given a state formula s and a number n, a bounded reachability query asks if any s-state is reachable from initial states in exactly n steps. The finite reachability engine answers these queries and provides a reason for the answer. In the case of reachability, the reason is a trace of $n + 1$ states leading from an initial state to an s-state. In the case of unreachability, the reason is an n-invariant blocking s.

The cooperation of these two engines is depicted in Figure 6.1. During the run, FiRE accumulates knowledge about the system in the form of bounded invariants. This knowledge helps it to answer subsequent queries faster. The progress of IcE during its run is modelled using a set of rules that capture and evolve the state of IcE. We discuss these rules in the next section.

The idea of separate components for inductive and bounded reachability reasoning is present already in [129]. However, our formalization enables us to easily extend the framework to the parallel setting with information sharing and reason about its correctness. In addition, thanks to its abstract nature, it covers not only PD-KIND [129] but also other algorithms, such as KIC3 [106]. In Section 6.3, we present PD-KIND as an instance of this framework in details.

6.2.1 Induction-Checking Engine

Given a safety problem for a transition system (I, T, P) the induction-checking engine (IcE) searches for k-inductive strengthening of P. It maintains two distinct sets of state formulas: a *base* frame $\mathcal{F}$ and a *successor* frame $\mathcal{G}$. In addition, it maintains information about its current level n. Intuitively, if IcE is currently working on level n, it already knows that the system is safe up to level n, i.e., $\neg P$ is not reachable in n steps or less. The base frame $\mathcal{F}$ serves both as a witness that $\neg P$ is not reachable, as well as a candidate for the inductive strengthening of P. IcE maintains an invariant that on level n every element of $\mathcal{F}$ is an n-invariant. Moreover, P is always an element of $\mathcal{F}$. The successor frame $\mathcal{G}$ collects those elements of $\mathcal{F}$ that are $\mathcal{F}^k$-inductive for some fixed $k \leq n+1$. Since $\bigwedge \mathcal{F}$ is an n-invariant, this means that all elements of $\mathcal{G}$ are at least $(n+1)$-invariants. When all elements of the base frame are checked and either successfully pushed to $\mathcal{G}$ or dropped, and no termination condition has been hit, $\mathcal{G}$ becomes the new base frame and the successor frame is emptied. If at any point $\mathcal{F} = \mathcal{G}$ then $\mathcal{F}$ is a k-inductive strengthening of P, proving that P holds in the system (as shown later in Lemma 6.1). In addition to the two frames IcE maintains a queue Q. The queue contains the elements of $\mathcal{F}$ that still need to be processed at the current level. We also refer to the elements of Q as *obligations*.

We now formalize the workings of the induction engine as a set of rules that work on and modify the current state of IcE. The current state of IcE is a 5-tuple $\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle$ with $\mathcal{F}$ being the base frame, $\mathcal{G}$ the successor frame, n the current level, Q the current queue of obligations, and k defining the current depth of induction. We refer to the state of IcE as *configuration.* For brevity we also sometimes refer to the elements of $\mathcal{F}$ as lemmas instead of bounded invariants. The initial configuration of IcE is $\langle \{P\}, \emptyset, 0, 1, \{P\} \rangle$ and IcE makes progress by applying the following rules. Note that the rules **Safe** and **Unsafe** are special, *terminating* rules.

Safe:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset \rangle}{SAFE} \qquad \textbf{if} \begin{cases} \mathcal{F} = \mathcal{G} \end{cases}$$

Unsafe:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle}{UNSAFE} \qquad \textbf{if} \begin{cases} \neg P \text{ is reachable in } [n+1, n+k] \text{ steps.} \end{cases}$$

Next-Level:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset \rangle}{\langle \mathcal{G}, \emptyset, n', k', \mathcal{G} \rangle} \qquad \textbf{if} \begin{cases} \mathcal{F} \neq \mathcal{G} \\ n' > n \\ \bigwedge \mathcal{G} \text{ is } n'\text{-invariant} \\ 1 \leq k' \leq n' + 1 \end{cases}$$

Push-Lemma:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathcal{F}, \mathcal{G} \cup \{l\}, n, k, Q \rangle} \qquad \textbf{if} \begin{cases} l \text{ is } \mathcal{F}^k\text{-inductive} \end{cases}$$

Add-Lemma:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle}{\langle \mathcal{F} \cup \{l\}, \mathcal{G}, n, k, Q \cup \{l\} \rangle} \qquad \textbf{if} \begin{cases} l \text{ is an } n\text{-invariant} \end{cases}$$

Drop-Lemma:

$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathcal{F}, \mathcal{G}, n, k, Q \rangle} \qquad \textbf{if} \begin{cases} l \neq P \end{cases}$$

The rules of IcE, namely **Add-Lemma** and **Drop-Lemma**, are abstract in the sense that we do not prescribe when or how are the new lemmas learnt, nor when they should be dropped. In sequential setting, new lemmas are typically learnt from FiRE when a counterexample to induction of some obligation is showed to be unreachable by FiRE. We discuss this in detail in Section 6.3 when we instantiate the abstract IcE for a concrete algorithm.

One specific thing that we would like to point out is that **Add-Lemma** is general enough to cover not only the *internal* learning, but also *external* learning. By internal learning we mean the learning of lemmas from FiRE. The external learning means that the lemmas can come from any other source. This is important for parallelization as it enables incorporating bounded invariants discovered by other instances working on the same problem.

Correctness of the induction-checking engine.

The abstract nature of the rules of IcE allows us to easily prove it correctness. That is, if the engine terminates by applying the rule **Safe** (**Unsafe**) then the system really is safe (unsafe).

Given our assumption that $I \implies P$, the following invariants are valid for the initial configuration and are maintained by every rule (excluding the terminating rules **Safe**, **Unsafe**):

1. $P \in \mathcal{F}$
2. For each $l \in \mathcal{F} \cup \mathcal{G} \cup Q$ at level n, l is an n-invariant of $\mathcal{S}$.
3. For each $l \in \mathcal{G}$, l is $\mathcal{F}^k$-inductive.

It is easy to verify that all invariants are valid for the initial configuration. The first invariant is trivially preserved by all rules except **Next-Level** as $\mathcal{F}$ either stays the same or grows. When **Next-Level** is applied that it must hold that $P \in \mathcal{G}$ since it is put in Q at the beginning of each level and can never be dropped. Since Q is empty when **Next-Level** is being applied, P must have been successfully pushed to $\mathcal{G}$ using **Push-Lemma**.

The second invariant is preserved by the rules **Next-Level**, **Push-Lemma** and **Drop-Lemma** since the set of formulas in consideration stays the same or becomes smaller. The invariant is also preserved by **Add-Lemma** because of the condition of the rule.

The third invariant trivially holds after applying **Next-Level** as the successor frame is empty at that moment. For the other rules, let us use $\mathcal{G}'$ to denote the successor frame after a rule has been applied. The invariant is also preserved by rules **Add-Lemma** and **Drop-Lemma** since $\mathcal{G}' = \mathcal{G}$. Finally, the invariant is preserved by **Push-Lemma** because of the condition of the rule.

Lemma 6.1. *When the algorithm terminates by applying **Safe**, the system satisfies the property P and $\bigwedge \mathcal{F}$ is a safe k-inductive invariant. When the algorithm terminates by applying **Unsafe**, the system can reach a state where P does not hold.*

Proof. The first part follows from the invariants. When **Safe** is applied, then it must be the case that $\mathcal{F} = \mathcal{G}$. This means that $\mathcal{F}$ is $\mathcal{F}^k$-inductive and consists of n-invariants of the system with $k \leq n + 1$. It follows that $\bigwedge \mathcal{F}$ is a k-inductive invariant of the system. Moreover, $P \in \mathcal{F}$, so P is an invariant. The second part follows trivially from the condition of the rule **Unsafe**. □

6.2.2 Finite Reachability Engine

The finite reachability engine (FiRE) is responsible for answering *bounded reachability queries* issued by IcE. A bounded reachability query for a system $\mathcal{S}$ is simply a pair $\langle s, i \rangle$ where s is a state formula and i is a natural number. It represents a question if any s-state is reachable in $\mathcal{S}$ by exactly i steps. This is naturally generalized to queries of the form $\langle s, [i, j] \rangle$, meaning reachability in at least i and at most j steps. An answer to a bounded reachability query $\langle s, i \rangle$ is either an i-invariant l such that $l \implies \neg s$ in case of unreachability, or a trace of $i + 1$ states starting from an initial state and ending in an s-state in case of reachability.

We do not prescribe how FiRE should be implemented, but we note two known instances: bounded model checking [29] and IC3/PDR [41]. An interesting observation [129] is that when IC3/PDR only needs to answer bounded reachability queries then the requirements on the frames it maintains can be relaxed. The frames do not need to be inductive nor form a monotone sequence.

From the parallelization perspective the advantage of FiRE based on bounded invariants is two-fold. First, the correctness of FiRE is maintained when bounded invariants are exchanged between different instances. Second, there is freedom in generalizing the bounded invariants computed as certificates of unreachability and this freedom can be exploited for portfolio approach to discover a variety of interesting bounded invariants across multiple instances.

6.2.3 Cooperation of Multiple Instance

We base our parallelization on the portfolio approach running multiple instances of the same algorithm with different parameters on a single problem. However, we aim to go beyond that. We want the instances to *cooperate* and to *share* information they discover about the problem they are solving. Our approach to cooperation of multiple instances of IcE/FiRE framework is depicted in Figure 6.2.

In our approach, several instances of IcE/FiRE framework (see Figure 6.1) work on the same problem and share information among themselves. However, the communication is split to that between the finite reachability engines and to that between induction-checking engines.

Cooperation of FiREs. Each reachability engine is gradually building and refining its representation of the state space by discovering and accumulating bounded invariants of the system. Since all instances work on the same transition system, a bounded invariant discovered by one instance is valid for other instances as well. Thus, multiple reachability engines can share their information through a global database of bounded invariants.

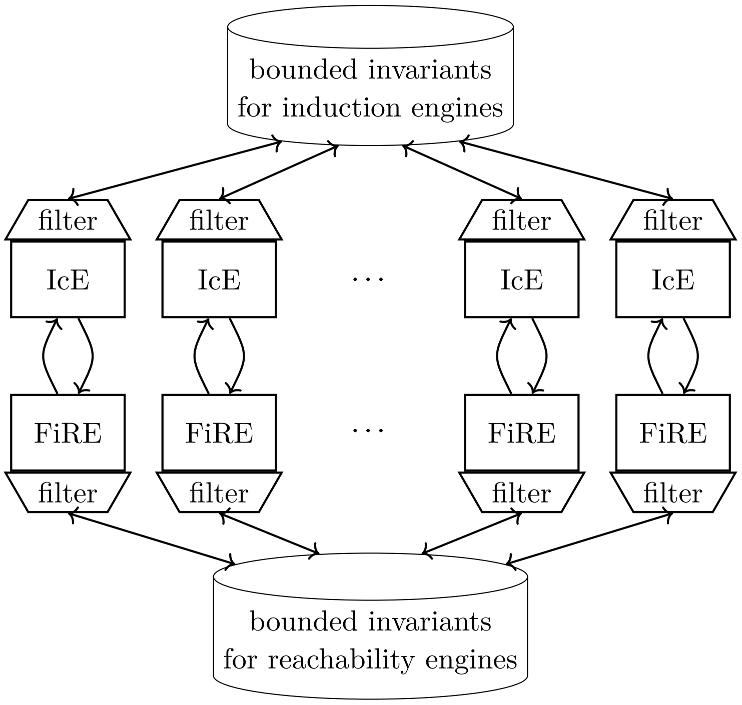


Figure 6.2. Multiple instances of IcE/FiRE framework sharing information

Additionally, in this setting each FiRE has a *filter* which controls which invariants are sent and received. The filter can be set to send and receive all or none invariants, or it can implement a heuristic. For example, it might be beneficial to send out only sufficiently small invariants to avoid burdening the other instances too much.

Cooperation of IcEs. Unlike FiREs, it is not immediately obvious what information IcEs could share between themselves. Natural candidates are elements of the base frame or the successor frame. However, one needs to be careful since different IcEs could be working on different levels and thus directly including lemmas from other instance might violate the invariants of these frames. Our solution is to accept external information in a way that can be modelled using the rule **Add-Lemma** and thus guarantee to preserve the correctness of the engine. Each engine sends out elements of the successor frame $\mathcal{G}$. When an engine is working on a level n and a lemma is pushed to $\mathcal{G}$, it is guaranteed to be at least $(n{+}1)$-invariant. Moreover, it is an *interesting* bounded invariant in the sense that this engine so far believes it should be part of the inductive strengthening. The engine sends such lemma to the global pool for other instances to see. When another engine receives this $(n{+}1)$-invariant, it checks if it can apply **Add-Lemma** to add it to its base frame. If the engine's current working level is higher than $n{+}1$, such bounded invariant cannot be added. Moreover, our preliminary experiments showed that it is better to have additional checks in the filter for incoming lemmas in order not to spend too much time processing useless external lemmas. We discuss our implementation and the experimental results with different settings of sharing information in Section 6.4.

6.3 PD-KIND as an Instance of IcE/FiRE

In this section, we reformulate the original description of PD-KIND [129] in terms of our IcE/FiRE framework. This reformulation enables us to identify the freedom in the algorithm that can be utilized for the portfolio approach to parallelization. Additionally, the techniques mentioned in Section 6.2 for sharing information between cooperating instances will become directly applicable for PD-KIND. On top of that, it allows us to prove the correctness of the parallel version of the algorithm.

6.3.1 Induction-Checking Engine of PD-KIND

The induction-checking engine of PD-KIND uses an extended configuration $\langle \mathcal{F}, \mathcal{G}, n, k, Q, n_{CTI} \rangle$, where n_{CTI} remembers the number of steps needed to reach a non-$\mathcal{F}$ state from an $\mathcal{F}$ state. This helps to determine $n' > n$ such that all elements of $\mathcal{G}$ are n'-invariants when applying **Next-Level**.

Additionally, IcE of PD-KIND maintains a mapping *CEX* of elements of $\mathcal{F}$ to potential counterexamples they block. Formally, *CEX* is a function from $\mathcal{F}$ to state formulas such that for each $l \in \mathcal{F}$, $l \implies \neg CEX(l)$ and every $CEX(l)$-state can reach a $\neg P$-state. Maintaining the potential counterexamples in addition to the bounded invariants allows for the earlier discovery of genuine counterexamples. It also provides a possible fallback in case the bounded invariant is too strong to be inductive.

The initial configuration of IcE is $\langle \{P\}, \emptyset, 0, 1, \{P\}, 1 \rangle$, with $CEX(P) = \neg P$. The engine makes progress using the following set of rules.

Safe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset, n_{CTI} \rangle}{SAFE} \quad \textbf{if} \left\{ \mathcal{F} = \mathcal{G} \right.$$

Next-Level:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, \emptyset, n_{CTI} \rangle}{\langle \mathcal{G}, \emptyset, n', k', \mathcal{G}, n' + k' \rangle} \quad \textbf{if} \begin{cases} \mathcal{F} \neq \mathcal{G} \\ n' = n + n_{CTI} \\ 1 \leq k' \leq n' + 1 \end{cases}$$

Push-Lemma:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F}, \mathcal{G} \cup \{l\}, n, k, Q, n_{CTI} \rangle} \quad \textbf{if} \left\{ l \text{ is } \mathcal{F}^k\text{-inductive} \right.$$

Unsafe:
$$\frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{UNSAFE} \quad \textbf{if} \left\{ CEX(l) \text{ is reachable in } [n{+}1, n{+}k] \text{ steps} \right.$$

$$\textbf{Add-Lemma:}\quad \frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G}, n, k, Q \cup \{l'\}, n_{CTI} \rangle} \quad \textbf{if} \begin{cases} \exists l \in Q \text{ s.t.} \\ \neg CEX(l) \text{ is not } \mathcal{F}^k\text{-inductive} \\ \text{with } c' \text{ being its } CTI \\ \textbf{Unsafe} \text{ is not applicable} \\ l' \text{ is } n\text{-invariant that blocks } c' \\ CEX(l') = c' \end{cases}$$

$$\textbf{Bad-Lemma:}\quad \frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G} \cup \{l'\}, n, k, Q, n'_{CTI})\rangle} \quad \textbf{if} \begin{cases} N \in [n{+}1, n{+}k] \\ \neg l \text{ reachable in } N \text{ steps} \\ l' = \neg CEX(l) \\ \neg CEX(l) \text{ is } \mathcal{F}^k\text{-inductive} \\ n'_{CTI} = min(N, n_{CTI}) \end{cases}$$

$$\textbf{Strengthen-Lemma:}\quad \frac{\langle \mathcal{F}, \mathcal{G}, n, k, Q \cup \{l\}, n_{CTI} \rangle}{\langle \mathcal{F} \cup \{l'\}, \mathcal{G}, n, k, Q \cup \{l'\}, n_{CTI} \rangle} \quad \textbf{if} \begin{cases} \neg CEX(l) \text{ is } \mathcal{F}^k\text{-inductive} \\ l \text{ is not } \mathcal{F}^k\text{-inductive} \\ \text{with } c' \text{ being } CTI \\ \textbf{Bad-Lemma} \text{ is not applicable} \\ l' \text{ is } n\text{-lemma s.t.} \\ l' \implies l \wedge \neg c' \\ CEX(l') = CEX(l) \end{cases}$$

A run of the engine starts from the initial configuration and applies the rules until **Safe** or **Unsafe** is applicable (which is generally not guaranteed to happen). The engine can be viewed as operating on a certain level, defined by the parameter n. At each level, the engine attempts to prove that the n-invariants from $\mathcal{F}$ are $\mathcal{F}^k$-inductive, strengthening the frame in the process if necessary or giving up on n-invariants that do not hold for higher levels. Two cases can happen when all elements of the (refined) frame $\mathcal{F}$ have been processed. Either the whole frame $\mathcal{F}$ has been pushed, in which case the engine can terminate using **Safe**, or some element could not be pushed, and thus **Next-Level** is applied.

If all elements have not been pushed yet, that is, Q is not empty, then an n-invariant l from Q is picked and processed in the following way: When l is $\mathcal{F}^k$-inductive then l, and consequently $\neg CEX(l)$, is in fact at least $(n{+}1)$-invariant. In this case, **Push-Lemma** is applied, and l is removed from Q.

If **Push-Lemma** is not applicable and $\neg CEX(l)$ is not $\mathcal{F}^k$-inductive, then there exists a *CTI* witnessing this. This *CTI* can be either real (reachable in $\mathcal{S}$) or spurious (not reachable in $\mathcal{S}$). A bounded reachability query is issued to FiRE to determine the status. If it is real, the system $\mathcal{S}$ is unsafe because $CEX(l)$ is reachable, and $\neg P$ is reachable from $CEX(l)$. In this case, the algorithm terminates by applying **Unsafe**. If *CTI* is spurious, then a new lemma blocking it is returned from FiRE and added to $\mathcal{F}$ by applying **Add-Lemma**.

The last possibility is that l is not $\mathcal{F}^k$-inductive, but $\neg CEX(l)$ is $\mathcal{F}^k$-inductive. Now the reachability query regarding the *CTI* for l is issued to FiRE. If it is not reachable, then l is strengthened using the reason of unreachability returned by FiRE – **Strengthen-Lemma** is applied. If it is reachable, then l is not an invariant of the system and must be discarded. **Bad-Lemma** is applied and l is replaced by $\neg \mathit{CEX}(l)$. Since we already know that $\neg \mathit{CEX}(l)$ is $\mathcal{F}^k$-inductive, it can be immediately pushed to the next frame.

This formalization of PD-KIND allows us to prove its correctness, building on the correctness of the abstract induction-checking engine (see Lemma 6.1). We extend the proof for the parallel version in Section 6.3.3.

Lemma 6.2. *If PD-KIND terminates using the rule* ***Safe (Unsafe)****, the transition system is safe (unsafe).*

Proof. For **Safe**, notice that PD-KIND's run can be viewed as a run of the abstract engine, as described in Section 6.2.1. To avoid name clashes, we use a prime to denote the PD-KIND's rules in this proof. All four rules **Safe**', **Push-Lemma**', **Next-Level**' and **Add-Lemma**' directly map to their abstract counterpart. **Bad-Lemma** is just **Drop-Lemma** applied on l followed by **Add-Lemma** and **Push-Lemma** on $\neg \mathit{CEX}(l)$. Finally, **Strengthen-Lemma** is **Drop-Lemma** applied on l, followed by **Add-Lemma** applied on l'. Consequently, each PD-KIND's run terminating with **Safe**' is mapped to the abstract engine's run terminating with **Safe**. By Lemma 6.1, the system is safe.

For **Unsafe**, we show that the following invariant is preserved throughout the run: For each l in $\mathcal{F} \cup \mathcal{G} \cup Q$, $\mathit{CEX}(l)$ can reach $\neg P$. The invariant holds for the initial configuration since $\mathcal{F} \cup \mathcal{G} \cup Q = \{P\}$ and $\mathit{CEX}(P) = \neg P$. **Add-Lemma** preserves the invariant since for the only new lemma l', $\mathit{CEX}(l')$ can reach $\mathit{CEX}(l)$, which can reach $\neg P$ by the induction hypothesis. The invariant is also preserved by **Bad-Lemma** and **Strengthen-Lemma** as $\mathit{CEX}(l') = \mathit{CEX}(l)$ for the only new lemma l' and the old lemma l. As the other rules do not change the set $\mathcal{F} \cup \mathcal{G} \cup Q$, we can conclude that the invariant is always preserved. Thus, when the algorithm terminates by rule **Unsafe**, $\neg P$ is reachable, and the system is unsafe. □

6.3.2 Finite Reachability Engine of PD-KIND

The finite reachability engine used in PD-KIND [129] can be described as an IC3-like algorithm. It answers bounded reachability queries using a sequence of *reachability frames* and local reasoning only, i.e., it does not unroll the transition relation. A *reachability frame* at level n, $\mathcal{R}_n$, is a set of n-invariants. Consequently, the set of $\mathcal{R}_n$-states over-approximates the set of states reachable in n steps or less. Unlike IC3, there is no further condition on the reachability frames. They do not need to be monotone nor form an inductive sequence. Like IC3, when FiRE receives a query $\langle s, i \rangle$, it checks if it is reachable in one step from $\mathcal{R}_{i-1}$ using a simple satisfiability query $\mathcal{R}_{i-1} \wedge T \wedge s'$. In the negative case, FiRE generalizes the reason for unreachability using *Craig interpolation* and reports the answer together with the reason. In the positive case, FiRE computes a predecessor t of s and recursively calls itself with query $\langle t, i{-}1 \rangle$. If this predecessor turns out to be unreachable, the $(i{-}1)$-invariant witnessing the unreachability is used to refine $\mathcal{R}_{i-1}$ and

s is rechecked. If the recursive sequence of calls ever reaches an initial state, then FiRE reports the query as reachable and returns the trace of the predecessors.

Note that the only requirement for the reachability frame $\mathcal{R}_n$ is that it contains only n-invariants. In the sequential setting, FiRE learns new bounded invariants on its own as it processes more and more reachability queries. However, in a parallel setting, it can also receive bounded invariants from an external source. More specifically, it can receive bounded invariants discovered by other instances of the same engine working in parallel on the same problem. Additionally, different interpolation algorithms can be used in different instances, thus allowing the engines to spread the search for useful bounded invariants.

6.3.3 Parallel PD-KIND

Since PD-KIND is an instantiation of the IcE/FiRE framework, it can be readily plugged into the abstract parallel framework with information sharing described in Section 6.2.3.

The bounded reachability information is stored in reachability frames consisting of bounded invariants. Whenever FiRE learns a new bounded invariant as a response to a bounded reachability query made by IcE, it can send it to the other instances. It can also periodically query the shared pool for new bounded invariants, and when it receives an external i-invariant, it can directly add it to its reachability frame $\mathcal{R}_i$.

Similarly, IcE sends out bounded invariants when it manages to push them to the successor frame. When it receives an external bounded invariant, it must check the necessary condition for adding it to the base frame. If the condition is not met, it simply drops the lemma. Otherwise, it uses a heuristic to determine the usefulness of the lemma. PD-KIND assumes that each base frame element is associated with a potential counterexample through the mapping CEX. Therefore, each bounded invariant l sent out by IcE must also be accompanied by its companion $CEX(l)$.

It is important for the success of a parallel approach to *diversify* the search for the solution. It was not possible to discuss this for the abstract framework as it requires the concrete algorithm with its concrete settings that drive the behaviour of the algorithm. Here we identify the key points where the behaviour of PD-KIND can be adjusted and finally give an algorithm capturing PD-KIND as an instance of IcE/FiRE framework in the parallel setting.

Choosing the depth of induction. When the induction engine moves to the next level n by applying **Next-Level**, there is freedom to choose a new value k of the induction depth from the interval $[1, n+1]$. The behaviour of the algorithm can be greatly influenced by the value of the induction depth it uses. For example, choosing large k requires a large unwinding of the transition relation when SAT/SMT solver is used and the inductive checks become slower. On the other hand, preferring larger k can lead to faster exploration of the search space. Moreover an obligation might be $\mathcal{F}^k$-inductive, and thus successfully pushed, but not $\mathcal{F}^{k'}$-inductive for $k' < k$. We denote the strategy to choose the new value of induction depth whenever **Next-Level** is applied as κ.

Obligation processing strategy. Several rules might be applicable given a configuration with a nonempty queue of obligations Q. However, once the obligation to be processed is chosen, there is no more freedom. The conditions of the rules are mutually exclusive for a fixed obligation $l \in Q$. Which rule applies for a particular obligation l is determined by its properties and the properties of $\mathit{CEX}(l)$. Therefore, the behaviour of the algorithm can be controlled through the strategy for picking the next obligation from the queue Q. We denote this strategy as ω.

Learning strategy. The finite reachability engine computes bounded invariants as certificates of unreachability. Theoretically, the certificate of unreachability for a query $\langle s, i\rangle$ could be $\neg s$. However, this leads to terrible performance in practice as it excludes only s and nothing else. Therefore, FiRE uses more sophisticated techniques to compute bounded invariants that are stronger and exclude more unreachable states. FiRE of PD-KIND uses Craig interpolation to compute bounded invariants. However, an interpolant for a given problem is generally not unique, and there exist techniques for computing different interpolants in propositional logic and theories of first-order logic. The use of different interpolation algorithms leads to different bounded invariants, which can significantly influence the performance of the whole algorithm (see Section 6.4). We denote the strategy for computing the bounded invariants as σ.

The run of a single instantiation of IcE/FiRE as PD-KIND in a parallel setting with information sharing is presented in pseudocode as Algorithm 6.1. The input is a triple $\mathcal{S} = \langle I, T, P\rangle$ representing the transition system and the property together with the three strategies κ, ω, σ that determine the behaviour of the algorithm at the previously identified non-deterministic steps.

Lemma 6.3. *The parallel version of PD-KIND with information exchange is correct. If it reports SAFE (UNSAFE), the system is safe (unsafe).*

Proof. The correctness of exchanging the bounded invariants between reachability engines has already been discussed in Section 6.2.3. The only new step IcE does, is incorporating an external lemma l from another PD-KIND instance, together with a potential counterexample that it blocks. An external lemma is learnt only if the condition of the abstract rule **Add-Lemma** is satisfied, and thus the invariants ensuring the correctness of the SAFE answer are preserved. Moreover, the invariant from the proof of Lemma 6.2 is preserved, and thus the UNSAFE answer is also correct. □

6.4 Implementation and Experiments

Our implementation of the parallel PD-KIND algorithm is based on the open-source model checker SALLY [129] and uses the SMTS framework [152] for parallelization and information exchange. We have extended SALLY with API for sending and receiving information. In our experiments SALLY was using YICES [83] for checking satisfiability

```
Procedure Run($\mathcal{S}, \kappa, \omega, \sigma$):
  $C = \langle \mathcal{F}, \mathcal{G}, n, k, Q, n_{CTI} \rangle \leftarrow \langle \{P\}, \emptyset, 0, 1, \{P\}, 1 \rangle$   // Initial configuration
  while $TRUE$ do
    if $Q = \emptyset$ then
      if $\mathcal{F} = \mathcal{G}$ then return $SAFE$        // Terminate using rule Safe
      else
        Apply Next-Level on $C$ with $\kappa$
        continue
    FiRE.SendReceive()          // FiRE sends and receives bounded invariants
    $C \leftarrow$ IcE.Receive($C$)          // IcE receives bounded invariants
    $l \leftarrow \omega(Q)$                    // Pick obligation to process
    $c \leftarrow CEX(l)$
    switch $\langle l, c \rangle$          // Pick rule based on properties of $l, c$
      case $l$ is $\mathcal{F}^k$-inductive
        Apply Push-Lemma for $l$ on $C$
        IcE.Send($\langle l, c, n+1 \rangle$)      // IcE sends pushed bounded invariant
      case $c$ is reachable in $[n+1, n+k]$ steps
        return $UNSAFE$                  // Terminate using rule Unsafe
      case $\neg c$ is not $\mathcal{F}^k$-inductive
        Apply Add-Lemma with $\sigma$ on $C$
      case $\neg l$ is reachable in $[n+1, n+k]$ steps
        Apply Bad-Lemma for $l$
      otherwise
        Apply Strengthen-Lemma with $\sigma$ on $C$ for $l$
```

Algorithm 6.1. PD-KIND in the parallel setting of IcE/FiRE

and OpenSMT [122] for the interpolation queries.[1]

The benchmarks were taken from the transition systems category of CHC COMP 2019[2], where the problem is encoded using the theory of linear real arithmetic. Out of 244 benchmarks, seven problematic ones were excluded due to the presence of a non-linear operation. All experiments were run on a single multi-core machine with 16 Intel® Xeon® X5687 @ 3.6 GHz CPUs and 180 GB of RAM. The resources were restricted to 1000 seconds of timeout and 6GB of memory *per one instance* of sally. This means that configurations with more instances are effectively granted more memory and CPU time. This choice is in line with our goal of improving the solver's wall clock time.

All instances use the default strategy of sally when they are choosing the depth

[1]All benchmarks, tools and results are bundled together in an artifact available at `https://doi.org/10.5281/zenodo.3484097`

[2]https://github.com/chc-comp/chc-comp19-benchmarks/tree/master/lra-ts

of induction (κ from Algorithm 6.1). The obligation processing strategy ω is a priority queue based on a score assigned to obligations, randomized to diversify the behaviour of different instances. The learning strategy σ is diversified primarily by using different interpolation algorithms in OpenSMT and secondary by using different random seeds for the SMT search. Three different LRA interpolation algorithms were used: Farkas interpolation algorithm [156], dual Farkas, and an interpolation algorithm based on decomposing Farkas interpolants [37]. We denote these as PF, DF and PD, respectively.

In the experiments, we seek answers to the following questions:

1. How does the system compare to the state-of-the-art?
2. How important is the sharing of information between various instances?
3. How does the approach scale when the number of instances is increased?
4. How do different interpolation algorithms contribute to the overall performance?

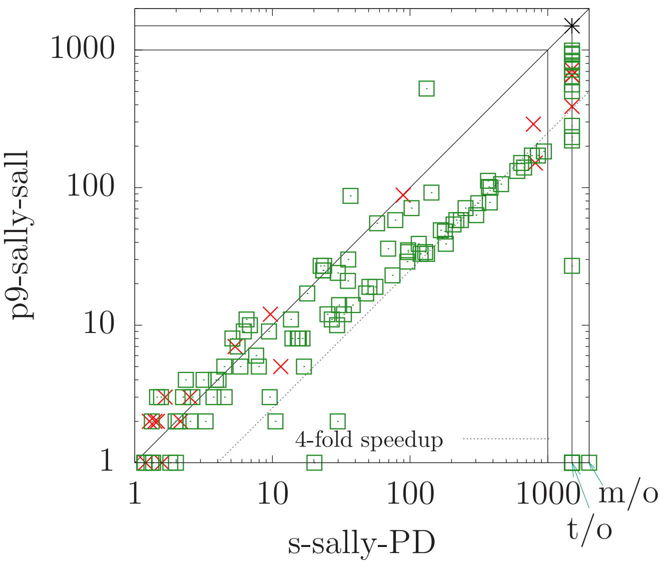


Figure 6.3. Best parallel configuration against the winner of LRA-TS category of CHC COMP 2019

Comparison to the state-of-the-art. The main result of the experiments is summarized in Figure 6.3 that compares the performance of the winner of the transition systems category of CHC COMP 2019 (sequential sally using PD interpolation algorithm in OpenSMT) with our parallel implementation with nine instances sharing information between IcEs and between FiREs. The parallel implementation achieves a 4-fold speedup on many instances and solves 224 instances compared to 197 instances solved by the sequential version.

We also compared our parallel implementation to P3 [150], the parallel implementation of Spacer [137] that also allows sharing information between solver instances. We also add the comparison with the sequential SPACER, the default Horn clause engine in Z3 [72].[3] The results are summarized in Figure 6.4. Our framework significantly outperforms SPACER on safe instances. Interestingly, SPACER fares better on unsafe instances.

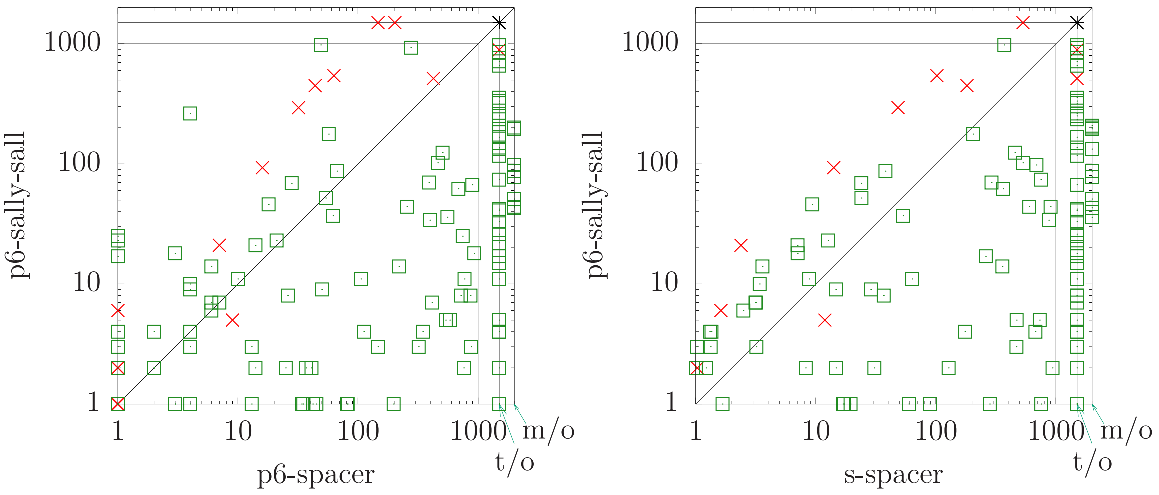


Figure 6.4. Comparison of parallel SALLY and parallel SPACER using 6 communicating instances

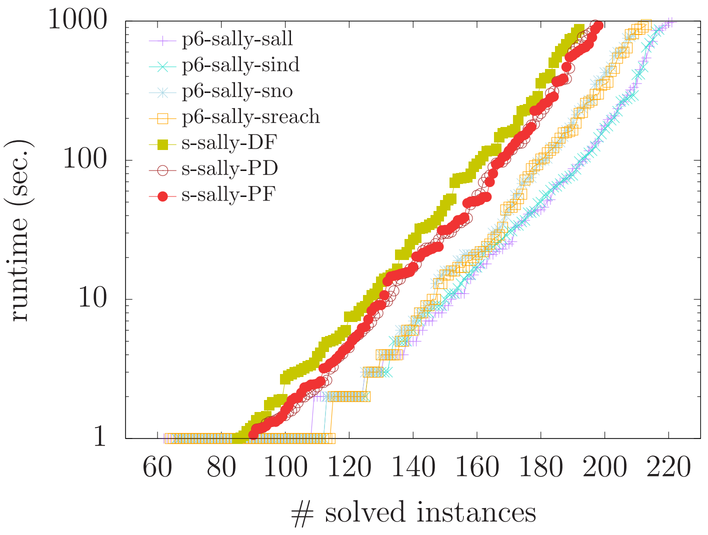


Figure 6.5. The effect of sharing information

Information sharing. Figure 6.5 summarizes the performance of 4 configurations: no information sharing (sno), sharing between FiREs only (sreach), sharing between IcEs only (sind), and all sharing enabled (sall). In these configurations, six instances ran in

[3]Results for Z3-4.8.5 with default settings.

parallel (two for each interpolation algorithm PF, DF and PD). For comparison, the figure includes results of sequential versions with different interpolation algorithms. Note that the runtimes of the parallel implementation were rounded to whole seconds, creating an effect of "stairs" for the low runtimes in cactus plots with a logarithmic scale. There is also a significant number of instances solved almost instantly; for this reason, the axes start at 1-second runtime and 50 instances solved.

A clear gap is visible between the best sequential version and the parallel versions, indicating that the parallel approach yields a significant improvement even without information sharing. Sharing information between FiREs is helpful, but the effect is not that significant compared to sharing information between IcEs, which is crucial for improving performance on many benchmarks. Configurations with sharing reachability information disabled (**p6-sally-sno**, **p6-sally-sind**) do not profit much from enabling it (**p6-sally-sreach**, **p6-sally-sall**). However, some hard benchmarks could only be solved by sharing reachability information. On the other hand, sharing the induction information boosts performance significantly. We conclude that the best performance was achieved by enabling sharing information between both IcEs and FiREs.

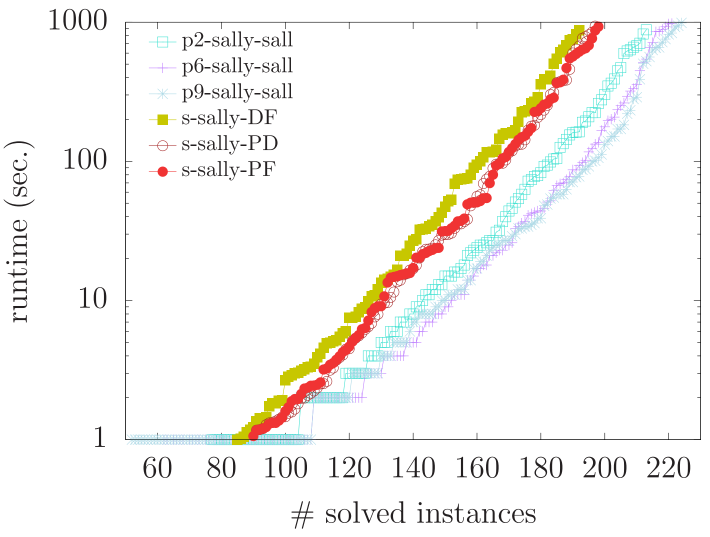


Figure 6.6. Scalability experiments

Scalability. We compared the performance of one, two, six and nine instances with all information sharing enabled. The results, summarized in Figure 6.6, show that adding more instances improves the performance, both decreasing the runtime and solving more benchmarks with the configurations solving 197, 213, 221 and 224 instances, respectively.

The effect of interpolation. The large jump when moving from sequential solving to two instances running in parallel can be in part attributed to different interpolation algorithms. We investigate this further in Figure 6.7. We compared configurations using six instances when the interpolation algorithm varies (**p6-sally-sall**), when the interpolation algorithm is fixed to PF for all instances (**p6-sally-sall-PF**), and when it is fixed to PD (**p6-sally-sall-PD**). We also added a configuration of just two instances (one with PF and one with PD). The results show that varying the interpolation algorithm is very important as the performance of **p2-sally-sall** is comparable to that of **p6-sally-sall-PD** and **p6-sally-sall-PF** while **p6-sally-sall** performs significantly better.

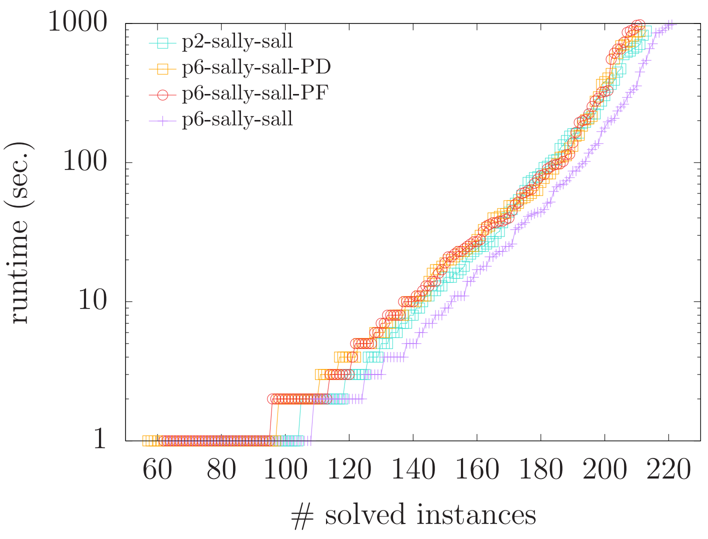


Figure 6.7. The effect of using different interpolation algorithms

The experiments show that our parallel algorithm performs substantially better than its sequential version. Its success can be attributed to more than one factor: The use of different interpolation algorithms helps to solve more benchmarks compared to a single interpolation algorithm used by all instances. Sharing information between solver instances can significantly reduce the runtime and thus solve more instances within the time limit. The major part of this can be attributed to the sharing of induction information, but sharing reachability information does help as well. The scalability experiments show continuing improvement up to nine instances. Additionally, our algorithm compares favorably with the state-of-the-art parallel implementation of Spacer, outperforming it significantly on the safe instances. Since Spacer is performing better on unsafe instances, the integration of the two algorithms within the SMTS framework to get the best of both tools is an interesting possibility for future work.

6.5 Related Work

Parallelization is a natural way of improving scalability of model-checking algorithms, for example, when facing the complexity of real-world problems. Below, we review the work that we deem most relevant to our results.

In [150], the authors presented the P3 system for parallelizing the IC3-inspired algorithm IC3/PDR for computing clusters using a portfolio of lemma-sharing solvers and search-space partitioning. The current work differs from that in several important aspects. First, we study a different class of algorithms, based on a combination of IC3 and k-induction. Second, in the implementation our emphasis in this work is on multicore environments instead of computing clusters. We also target a different application domain, studying transition systems instead of general constrained Horn clauses. Finally, in comparing the current system against P3 we measure a significant improvement on the set of instances that both tools can solve, providing practical evidence on the importance of the contribution.

Approaches for parallel IC3 were suggested, for example, in the original publication [41], and more recently in [50]. The current system differs from both, in addition to basing on k-induction, by allowing constraints expressible in first-order logic through an SMT encoding instead of purely propositional encoding, therefore being more readily applicable in software model checking.

The Tarmo system [198] allows SAT-based bounded model checkers to share learned clauses between queries of different execution bounds. The approach could be applied at least in the FiRE systems underlying our bounded reachability queries by allowing the SMT solvers to share clauses as in [123, 151]. However, we leave the study of the performance effects of such a technique for future work.

A system presented in [168] follows a different approach to determining the feasibility of symbolic execution paths in parallel. Our approach is more symbolic in the sense that it does not require the explicit enumeration of, in general, an exponential number of paths done in [168]. Algorithms for parallel LTL model checking are presented in [13]. The general approach relies on an automata-theoretic formulation of reducing model checking to determining the emptiness of Büchi automata. The parallelization idea focuses on using algorithms based on DFS and BFS for this purpose. We consider this approach orthogonal to ours and leave it for future work to study the possible synergies. In [133], the authors use three processes to parallelize a standard k-induction algorithm enriched with invariants generated from predefined templates. This approach was generalized in [21] where program analysis with dynamic precision refinement generates continuously-refined invariants for the k-induction. Our parallelization approach is based on a more general framework, and allows scalability to arbitrary number of cores. In [162], the authors present a more general approach of parallelizing model checking by running several model checkers in parallel. However, the paper does not address the problem of sharing information between the solvers, a topic central to our work.

Finally, our approach is greatly inspired by the sequential approaches combining k-induction with IC3, in particular the PD-KIND algorithm [129] but also the KIC3 framework [106]. The aim of IcE/FiRE is to capture the class of these algorithms from the point of view of information sharing between different solvers, and apply these results to parallelize these algorithms.

A recent work [20] presents another approach of combining k-induction and IC3/PDR.

It extends the framework of [21] and employs IC3/PDR (not only) for the generation of auxiliary invariants for k-induction.

Combining and unifying different approaches to software verification, such as IC3/PDR [41, 85], k-induction [186] and BMC [29], is becoming increasingly popular [21, 22, 43, 106, 129]. Both combination and parallelization techniques benefit from relentless continuous improvements [31, 58, 113, 144, 193] of the original algorithms.

6.6 Conclusions and Future Work

In this chapter, we contributed IcE/FiRE framework, which generalizes the concepts from a recently developed class of algorithms combining IC3 and k-induction, with PD-KIND being the prominent representative. The architecture consists of separate components for inductive reasoning and bounded reachability, which is particularly suitable for exchanging learnt information in a parallel setting.

Using the setting of IcE/FiRE framework, we have derived a parallel version of PD-KIND algorithm and implemented it in SMTS framework for distributed solving. We show experimentally that this approach provides a good speed-up in multi-core environments; the parallel solver surpasses the current state-of-the-art in proving safety of transitions systems both in speed and the number of instances solved.

As a future direction, an interesting line of research would be to incorporate techniques for search space partitioning and study closer possible heuristics for sharing lemmas between the solvers. It remains an open question how to extend the framework to capture not only algorithms that learn information about *states*, but also algorithms that learn information about *transitions*, such as the TPA algorithm from Chapter 4.

Chapter 7

Conclusions

This thesis studied the problem of *automated formal verification*, a task to automatically prove, in the mathematical sense, that a system satisfies its specification, or find a behaviour of the system that violates the specification. We chose to study the task in the modelling framework of *constrained Horn clauses* (CHC) and its particular fragment represented by *symbolic transition systems*. The logical representation of this modelling language enables simpler formal reasoning about the problem and proposed solving techniques, as well as direct integration with powerful SMT solvers as the underlying reasoning engines.

The main message of this thesis is that formal verification is a complex task consisting of more than one layer, and a successful approach to solving this task requires deep knowledge of the whole stack. We divide the task into *foundational*, *verification*, and *cooperative* layers, which correspond to decision and interpolation procedures implemented in SMT solvers, SMT-based model checking algorithms and multi-agent parallel solving, respectively. Although SMT solvers can be used as black boxes in model checking and software verification algorithms, we argue that true efficiency is obtained only with tight integration of the verifier with the underlying SMT solver as the foundational component. An efficient integration requires full utilization of the strengths of the SMT solver and avoidance of its weaknesses, which is possible only with an understanding of its inner workings. Similarly, designing a parallel solving algorithm in a multi-agent setting requires intimate knowledge of the single solver instance, including its foundational component. Awareness of the essential parameters, such as an interpolation algorithm, and how to tune them can significantly affect the performance already in a simple parallel portfolio. However, cooperation in the form of information exchange between agents is required to achieve truly scalable performance. The decisions of what information can be exchanged and how the sequential agent can integrate the external information require an understanding of the underlying model-checking algorithm.

In Chapter 3, we proposed a way to decompose Farkas interpolants using techniques from linear algebra. If such decomposition is possible, this yields a logically stronger interpolant in the form of a conjunction of inequalities. In the SMT setting where interpolants are computed from proofs of unsatisfiability, logically stronger interpolants for theory conflicts yield a logically stronger interpolant for the overall SMT problem. This novel interpolation algorithm is successfully applied in the rest of our work. In the multi-agent setting in Chapter 6, using multiple interpolation algorithms results in more diverse behaviour across a group of agents and leads to better performance of our implementation of the parallel PD-KIND algorithm. The TPA algorithm given in Chapter 4 heavily relies on interpolants, and decomposed Farkas interpolants are used in our implementation, as adding them improved performance.

In Chapter 4 we focused on the verification layer and scalability problem of existing model checking techniques on systems with faulty behaviour that requires a long time to manifest. In the language of model checking, these are transition systems with only a very long counterexample path. Based on our experience with existing techniques and their weaknesses, we developed a concept of *transition power abstraction sequence* and a model-checking algorithm based on this concept. This algorithm is relatively simple yet powerful, as we have experimentally demonstrated on problems representing challenging multi-phase loops. Additionally, we showed that TPA sequence can also be used to prove safety by discovering safe inductive *transition* invariant. Since the original algorithm was not sufficiently effective in proving safety, we developed the concept further and introduced SPLIT-TPA. SPLIT-TPA is based on the idea of splitting the original TPA sequence into two components. The splitting introduced more candidates for transition invariants and enabled the application of efficient k-inductive reasoning. The experimental evaluation confirmed that SPLIT-TPA dominates TPA in proving safety while retaining the ability to detect deep counterexamples. Additionally, it can prove safe some problems not solvable by other state-of-the-art techniques, demonstrating the usefulness of transition invariants as a proof rule.

In Chapter 5 we described the Horn solver GOLEM that we developed as part of our research. GOLEM has been developed to implement and study interpolation-based (and other SMT-based) model-checking algorithms, providing infrastructure and efficiency by tight integration with the underlying SMT solver OPENSMT. The comparison with other Horn solvers in the latest edition of CHC-COMP[1] shows that GOLEM has great potential to join the ranks of SPACER and ELDARICA as the go-to back-end solver for the growing number of applications that model their problems using the language of *constrained Horn clauses* and rely on off-the-shelf Horn solvers for solving. Besides the use as the back-end solver for domain-specific tools, GOLEM can also drive the research at the foundational layer, as different engines of GOLEM stress various aspects of SMT solving and related procedures of interpolation and model-based projection. This leads to new challenges, especially for richer SMT theories, including arrays, bit-vectors and algebraic data types.

[1] GOLEM outperformed all other solvers except SPACER in LRA-TS, LIA-lin and LIA-nonlin tracks, which were all the tracks where GOLEM participated.

In Chapter 6, we proposed the IcE/FiRE framework, which abstracts concepts from a recently developed class of algorithms based on a combination of IC3 and k-induction. IcE/FiRE was explicitly designed with the application in a multi-agent setting with information exchange in mind. Instantiating the parallel IcE/FiRE framework with PD-KIND algorithm [129], we obtained a parallel PD-KIND solver that showed substantial improvements over the sequential version, as well as over another parallel model checker P3 [150].

In conclusion, this thesis presents our work that successfully advanced the state-of-the-art in automated verification. By taking a layered approach to the general verification problem, we made contributions to the foundational techniques with a new interpolation algorithm for LRA; to model checking with the concept of transition power abstraction sequence and algorithms TPA and SPLIT-TPA; to the application of parallel multi-agent solving in model checking with IcE/FiRE framework and parallel PD-KIND that instantiates this framework; and to the tool support with our Horn solver GOLEM. We hope our work encourages more researchers to pursue deeper knowledge of the components they work with rather than treating them as black boxes. Deep knowledge is especially powerful for using SMT solvers in verification, but also for using Horn solvers in domain-specific verification tools and in multi-agent settings. The awareness of the strengths and weaknesses of the underlying solvers should guide the modelling part of the verification task. Often there are multiple ways of encoding a domain-specific verification task, and the chosen approach can significantly help or hinder the underlying solver.

Future work

Several directions for future research can build on the work presented in this thesis. Regarding interpolation-based model-checking algorithms, where interpolant computation is a procedure invoked frequently, choosing the best interpolation procedure for each interpolation task remains an open question.

Parallelization and multi-agent approaches to verification represent a research direction on their own. While applying a portfolio approach is typically straightforward, more sophisticated techniques like partitioning and information exchange are much more challenging. While partitioning techniques in the context of CHC can be general, information exchange depends on the particular back-end algorithm. In the context of parallel PD-KIND, we investigated the exchange of *positive* information, i.e., bounded invariants of the system that are likely to participate in the final invariant. However, it might be beneficial to exchange also *negative* information. For example, the information that certain bounded invariant has been proven not to be unbounded and thus cannot participate in the final inductive invariant.

We believe the concept of *transition power abstraction* offers many opportunities for further study. The algorithms TPA and SPLIT-TPA utilize this concept for detecting deep counterexamples and discovering safe transition invariants. However, we see much more potential here. There are variants of the algorithms that could be more efficient, more suitable for a particular class of problems, or more suitable for generalization to nonlinear Horn-clause systems. The possibility of connecting the TPA approach with Spacer-like search is especially intriguing. Additionally, the automatic discovery of transition invariants in TPA could find new applications in the analysis of liveness properties and analysis of termination or non-termination. Another exciting possibility is to investigate if TPA can discover deep counterexamples in the hardware model checking domain, with systems modelled at the word level, using the SMT theory of bit-vectors.

Golem also offers many different possibilities, as we outlined in Section 5.8, and we plan to continue developing the tool. To make Golem more attractive for domain-specific verification tools, the support of more background theories, including arrays, bit-vectors and algebraic data types, is desired. This requires mainly the support of the theories in OpenSMT, but also offers research opportunities regarding new procedures for interpolation and model-based projection. Implementing additional preprocessing transformations and extending the set of back-end engines will also make Golem more powerful. Implementing existing (and potentially new) algorithms in the same tool facilitates their understanding and comparison. This is important for new research ideas that address the weaknesses of existing techniques while drawing inspiration from their strengths.

Bibliography

[1] Abiteboul, S., Hull, R. and Vianu, V. [1995]. *Foundations of Databases*, Addison-Wesley.
URL: *http://webdam.inria.fr/Alice/*

[2] Albarghouthi, A., Gurfinkel, A. and Chechik, M. [2012]. Whale: An interpolation-based algorithm for inter-procedural verification, *Verification, Model Checking, and Abstract Interpretation (VMCAI '12)*, Vol. 7148 of *LNCS*, pp. 39–55.

[3] Albarghouthi, A. and McMillan, K. L. [2013]. Beautiful interpolants, *in* N. Sharygina and H. Veith (eds), *CAV 2013*, Vol. 8044 of *LNCS*, Springer, Heidelberg, pp. 313–329.

[4] Alt, L. [2016]. *Controlled and Effective Interpolation*, PhD thesis, Università della Svizzera italiana.

[5] 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*, Springer International Publishing, Cham, pp. 325–338.

[6] Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2016]. A proof-sensitive approach for small propositional interpolants, *in* A. Gurfinkel and S. A. Seshia (eds), *VSTTE 2015*, Vol. 9593 of *LNCS*, Springer, Cham, pp. 1–18.

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

[8] 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 Software: Verification and Testing*, Springer International Publishing, Cham, pp. 195–210.

[9] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E. and Udupa, A. [2013]. Syntax-guided synthesis, *2013 Formal Methods in Computer-Aided Design*, pp. 1–8.

[10] Andrilli, S. and Hecker, D. [2016]. *Elementary Linear Algebra*, 5 edn, Academic Press.

[11] Barbosa, H., Barrett, C., 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*, Springer International Publishing, Cham, pp. 415–442.

[12] Bardin, S., Finkel, A., Leroux, J. and Petrucci, L. [2008]. FAST: Acceleration from theory to practice, *International Journal on Software Tools for Technology Transfer* **10**(5): 401–424.

[13] Barnat, J., Bloemen, V., Duret-Lutz, A., Laarman, A., Petrucci, L., van de Pol, J. and Renault, E. [2018]. *Parallel Model Checking Algorithms for Linear-Time Temporal Logic*, Springer International Publishing, Cham, pp. 457–507.

[14] Barrett, C., de Moura, L., Ranise, S., Stump, A. and Tinelli, C. [2011]. The SMT-LIB initiative and the rise of SMT, *in* S. Barner, I. Harris, D. Kroening and O. Raz (eds), *Hardware and Software: Verification and Testing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 3–3.

[15] Barrett, C., Fontaine, P. and Tinelli, C. [2016]. The Satisfiability Modulo Theories Library (SMT-LIB). Accessed on 20.12.2022.
URL: *https://smtlib.cs.uiowa.edu/*

[16] Barrett, C., Fontaine, P. and Tinelli, C. [2017]. The SMT-LIB Standard: Version 2.6, *Technical report*, Department of Computer Science, The University of Iowa.

[17] Barrett, C., Sebastiani, R., Seshia, S. and Tinelli, C. [2009]. *Satisfiability modulo theories*, Vol. 185 of *Frontiers in Artificial Intelligence and Applications*, 1 edn, pp. 825–885.

[18] Berryhill, R., Ivrii, A., Veira, N. and Veneris, A. [2017]. Learning support sets in IC3 and Quip: The good, the bad, and the ugly, *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 140–147.

[19] Beyer, D. [2022]. Progress on software verification: SV-COMP 2022, *in* D. Fisman and G. Rosu (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 375–402.

[20] Beyer, D. and Dangl, M. [2020]. Software verification with PDR: An implementation of the state of the art, *in* A. Biere and D. Parker (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 3–21.

[21] Beyer, D., Dangl, M. and Wendler, P. [2015]. Boosting k-induction with continuously-refined invariants, *in* D. Kroening and C. S. Păsăreanu (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 622–640.

[22] Beyer, D., Dangl, M. and Wendler, P. [2018]. A unifying view on SMT-based software verification, *Journal of Automated Reasoning* **60**(3): 299–335.

[23] Beyer, D., Henzinger, T. A. and Théoduloz, G. [2007]. Configurable software verification: Concretizing the convergence of model checking and program analysis, *in* W. Damm and H. Hermanns (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 504–518.

[24] Beyer, D., Kanav, S. and Richter, C. [2022]. Construction of verifier combinations based on off-the-shelf verifiers, *in* E. B. Johnsen and M. Wimmer (eds), *Fundamental Approaches to Software Engineering*, Springer International Publishing, Cham, pp. 49–70.

[25] Beyer, D. and Keremoglu, M. E. [2011]. CPAchecker: A tool for configurable software verification, *in* G. Gopalakrishnan and S. Qadeer (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 184–190.

[26] Beyer, D., Lee, N.-Z. and Wendler, P. [2022]. Interpolation and SAT-based model checking revisited: Adoption to software verification, *Technical Report 2208.05046*, arXiv/CoRR.
URL: *https://www.sosy-lab.org/research/cpa-imc/*

[27] Beyer, D. and Wehrheim, H. [2020]. Verification artifacts in cooperative verification: Survey and unifying component framework, *in* T. Margaria and B. Steffen (eds), *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, Springer International Publishing, Cham, pp. 143–167.

[28] Beyer, D. and Wendler, P. [2012]. Algorithms for software model checking: Predicate abstraction vs. Impact, *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 106–113.

[29] Biere, A., Cimatti, A., Clarke, E. and Zhu, Y. [1999]. Symbolic model checking without BDDs, *in* W. R. Cleaveland (ed.), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 193–207.

[30] Biere, A., Heule, M., van Maaren, H. and Walsh, T. [2009]. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*, IOS Press, NLD.

[31] Birgmeier, J., Bradley, A. R. and Weissenbacher, G. [2014]. Counterexample to induction-guided abstraction-refinement (CTIGAR), *in* A. Biere and R. Bloem (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 831–848.

[32] Bjørner, N., Gurfinkel, A., McMillan, K. and Rybalchenko, A. [2015]. Horn clause solvers for program verification, *Lecture Notes in Computer Science* pp. 24–51.

[33] Bjørner, N. and Janota, M. [2015]. Playing with quantified satisfaction, *in* A. Fehnker, A. McIver, G. Sutcliffe and A. Voronkov (eds), *LPAR-20. 20th International Conferences on Logic for Programming, Artificial Intelligence and Reasoning - Short Presentations*, Vol. 35 of *EPiC Series in Computing*, EasyChair, pp. 15–27.

[34] Bjørner, N. and Nachmanson, L. [2020]. Navigating the universe of Z3 theory solvers, *in* G. Carvalho and V. Stolz (eds), *Formal Methods: Foundations and Applications*, Springer International Publishing, Cham, pp. 8–24.

[35] Blicha, M., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2022a]. Split transition power abstractions for unbounded safety, *in* A. Griggio and N. Rungta (eds), *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*, TU Wien Academic Press, pp. 349–358.

[36] 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*, Springer International Publishing, Cham, pp. 524–542.

[37] Blicha, M., Hyvärinen, A. E. J., Kofroň, 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*, Springer International Publishing, Cham, pp. 3–20.

[38] Blicha, M., Hyvärinen, A. E. J., Kofroň, J. and Sharygina, N. [2022]. Using linear algebra in decomposition of Farkas interpolants, *International Journal on Software Tools for Technology Transfer* **24**(1): 111–125.

[39] Blicha, M., Hyvärinen, A. E. J., Marescotti, M. and Sharygina, N. [2020]. A cooperative parallelization approach for property-directed k-induction, *in* D. Beyer and D. Zufferey (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer International Publishing, Cham, pp. 270–292.

[40] Bozga, M., Iosif, R. and Konečný, F. [2010]. Fast acceleration of ultimately periodic relations, *in* T. Touili, B. Cook and P. Jackson (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 227–242.

[41] Bradley, A. R. [2011]. SAT-based model checking without unrolling, *in* R. Jhala and D. Schmidt (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 70–87.

[42] Bradley, A. R. [2012]. Understanding IC3, *in* A. Cimatti and R. Sebastiani (eds), *Theory and Applications of Satisfiability Testing – SAT 2012*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–14.

[43] Brain, M., Joshi, S., Kroening, D. and Schrammel, P. [2015]. Safety verification and refutation by k-invariants and k-induction, *in* S. Blazy and T. Jensen (eds), *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 145–161.

[44] Bromberger, M. [2019]. *Decision Procedures for Linear Arithmetic*, PhD thesis, Saarland University.

[45] Bruttomesso, R., Ghilardi, S. and Ranise, S. [2012]. Quantifier-free interpolation of a theory of arrays, *Logical Methods in Computer Science* **8**(2).

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

[47] Bueno, D. and Sakallah, K. A. [2020]. Horn2VMT: Translating Horn reachability into transition systems, *7th Workshop on Horn Clauses for Verification and Synthesis (HCVS)*, Dublin, Ireland.

[48] Calzavara, S., Grishchenko, I. and Maffei, M. [2016]. HornDroid: Practical and sound static analysis of android applications by SMT solving, *2016 IEEE European Symposium on Security and Privacy*, pp. 47–62.

[49] Caniart, N., Fleury, E., Leroux, J. and Zeitoun, M. [2008]. Accelerating interpolation-based model-checking, *in* C. R. Ramakrishnan and J. Rehof (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 428–442.

[50] Chaki, S. and Karimi, D. [2016]. Model checking with multi-threaded IC3 portfolios, *in* B. Jobstmann and K. R. M. Leino (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 517–535.

[51] Champion, A., Chiba, T., Kobayashi, N. and Sato, R. [2018]. ICE-based refinement type discovery for higher-order functional programs, *in* D. Beyer and M. Huisman (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer International Publishing, Cham, pp. 365–384.

[52] Champion, A., Kobayashi, N. and Sato, R. [2018]. HoIce: An ICE-based non-linear Horn clause solver, *in* S. Ryu (ed.), *Programming Languages and Systems*, Springer International Publishing, Cham, pp. 146–156.

[53] Chong, N., Cook, B., Kallas, K., Khazem, K., Monteiro, F. R., Schwartz-Narbonne, D., Tasiran, S., Tautschnig, M. and Tuttle, M. R. [2020]. Code-level model checking

in the software development workflow, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, Association for Computing Machinery, New York, NY, USA, pp. 11–20.
URL: *https://doi.org/10.1145/3377813.3381347*

[54] Christ, J. and Hoenicke, J. [2015]. Weakly equivalent arrays, *in* C. Lutz and S. Ranise (eds), *Frontiers of Combining Systems*, Springer International Publishing, Cham, pp. 119–134.

[55] Christ, J. and Hoenicke, J. [2016]. Proof tree preserving tree interpolation, *Journal of Automated Reasoning* **57**(1): 67–95.

[56] Christ, J., Hoenicke, J. and Nutz, A. [2012]. SMTInterpol: An interpolating SMT solver, *in* A. Donaldson and D. Parker (eds), *Model Checking Software*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 248–254.

[57] Christ, J., Hoenicke, J. and Nutz, A. [2013]. Proof tree preserving interpolation, *in* N. Piterman and S. A. Smolka (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 124–138.

[58] Cimatti, A. and Griggio, A. [2012]. Software model checking via IC3, *in* P. Madhusudan and S. A. Seshia (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 277–293.

[59] Cimatti, A., Griggio, A., Mover, S. and Tonetta, S. [2014]. IC3 modulo theories via implicit predicate abstraction, *in* E. Ábrahám and K. Havelund (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 46–61.

[60] Cimatti, A., Griggio, A., Schaafsma, B. and Sebastiani, R. [2013]. The MathSAT5 SMT solver, *in* N. Piterman and S. Smolka (eds), *TACAS 2013*, Vol. 7795 of *LNCS*, Springer.

[61] Cimatti, A., Griggio, A. and Sebastiani, R. [2010]. Efficient generation of Craig interpolants in satisfiability modulo theories, *ACM Trans. Comput. Logic* **12**(1): 7:1–7:54.

[62] Cimatti, A., Griggio, A. and Tonetta, S. [2021]. The VMT-LIB language and tools.

[63] Clarke, E., 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*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 154–169.

[64] Clarke, E. M. and Emerson, E. A. [1982]. Design and synthesis of synchronization skeletons using branching time temporal logic, *in* D. Kozen (ed.), *Logics of Programs*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 52–71.

[65] *CoCoSim* [n.d.]. `https://coco-team.github.io/cocosim`.

[66] Cook, S. A. [1971]. The complexity of theorem-proving procedures, *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, Association for Computing Machinery, New York, NY, USA, pp. 151–158.
URL: *https://doi.org/10.1145/800157.805047*

[67] Cousot, P. and Cousot, R. [1977]. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, Association for Computing Machinery, New York, NY, USA, pp. 238–252.

[68] 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.

[69] Daniel, J., Cimatti, A., Griggio, A., Tonetta, S. and Mover, S. [2016]. Infinite-state liveness-to-safety via implicit abstraction and well-founded relations, *in* S. Chaudhuri and A. Farzan (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 271–291.

[70] Davis, M. [1958]. *Computability and unsolvability*, McGraw-Hill.

[71] De Angelis, E. and Vediramana Krishnan, H. G. [2022]. CHC-COMP 2022: Competition report, *Electronic Proceedings in Theoretical Computer Science* **373**: 44–62.

[72] de Moura, L. and Bjørner, N. [2008]. Z3: An efficient SMT solver, *in* C. R. Ramakrishnan and J. Rehof (eds), *TACAS 2008*, Springer, Heidelberg, pp. 337–340.

[73] de Moura, L., Rueß, H. and Sorea, M. [2003]. Bounded model checking and induction: From refutation to verification, *in* W. A. Hunt and F. Somenzi (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 14–26.

[74] Detlefs, D., Nelson, G. and Saxe, J. B. [2005]. Simplify: A theorem prover for program checking, *Journal of the ACM* **52**(3): 365–473.

[75] Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A. and Podelski, A. [2019]. Ultimate TreeAutomizer (CHC-COMP tool description), *in* E. D. Angelis, G. Fedyukovich, N. Tzevelekos and M. Ulbrich (eds), *Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019*, Vol. 296 of *EPTCS*, pp. 42–47.

[76] Dijkstra, E. W. [1970]. Notes on structured programming.

[77] Dijkstra, E. W. [1972]. The humble programmer, *Commun. ACM* **15**(10): 859–866.
URL: *https://doi.org/10.1145/355604.361591*

[78] Dillig, I., Dillig, T. and Aiken, A. [2009]. Cuts from proofs: A complete and practical technique for solving linear inequalities over integers, *in* A. Bouajjani and O. Maler (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 233–247.

[79] Donaldson, A. F., Haller, L., Kroening, D. and Rümmer, P. [2011]. Software verification using k-induction, *in* E. Yahav (ed.), *Static Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 351–368.

[80] Donaldson, A. F., Kroening, D. and Rümmer, P. [2011]. Automatic analysis of DMA races using model checking and k-induction, *Formal Methods in System Design* **39**(1): 83–113.

[81] D'Silva, V., Kroening, D., Purandare, M. and Weissenbacher, G. [2010]. Interpolant strength, *VMCAI 2010*, Vol. 5944 of *LNCS*, Springer, pp. 129–145.

[82] Dureja, R., Gurfinkel, A., Ivrii, A. and Vizel, Y. [2021]. IC3 with internal signals, *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design – FMCAD 2021*, TU Wien Academic Press, pp. 63–71.

[83] Dutertre, B. [2014]. Yices 2.2, *in* A. Biere and R. Bloem (eds), *Computer-Aided Verification (CAV'2014)*, Vol. 8559 of *Lecture Notes in Computer Science*, Springer, pp. 737–744.

[84] Dutertre, B. and de Moura, L. M. [2006]. A fast linear-arithmetic solver for DPLL(T), *in* T. Ball and R. B. Jones (eds), *CAV 2006*, Vol. 4144 of *LNCS*, Springer, pp. 81–94.

[85] Een, N., Mishchenko, A. and Brayton, R. [2011]. Efficient implementation of property directed reachability, *Proceedings of the International Conference on Formal Methods in Computer-Aided Design*, FMCAD '11, FMCAD Inc, Austin, TX, pp. 125–134.

[86] Eén, N. and Sörensson, N. [2004]. An extensible SAT-solver, *in* E. Giunchiglia and A. Tacchella (eds), *Theory and Applications of Satisfiability Testing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 502–518.

[87] Ernst, G. [2020]. A complete approach to loop verification with invariants and summaries.

[88] Esen, Z. and Rümmer, P. [2022]. TriCera: Verifying C programs using the theory of heaps, *in* A. Griggio and N. Rungta (eds), *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*, TU Wien Academic Press, pp. 360–391.

[89] Faella, M. and Parlato, G. [2022]. Reasoning about data trees using CHCs, *in* S. Shoham and Y. Vizel (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 249–271.

[90] Farkas, G. [1894]. A Fourier-féle mechanikai elv alkalmazásai (Hungarian) [On the applications of the mechanical principle of Fourier].

[91] Fedyukovich, G. and Bodík, R. [2018]. Accelerating syntax-guided invariant synthesis, *TACAS, Part I*, Vol. 10805 of *LNCS*, Springer, pp. 251–269.

[92] Fedyukovich, G., Kaufman, S. J. and Bodík, R. [2017]. Sampling invariants from frequency distributions, *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 100–107.

[93] Fedyukovich, G., Prabhu, S., Madhukar, K. and Gupta, A. [2018]. Solving constrained Horn clauses using syntax and data, *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–9.

[94] 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*, Springer International Publishing, Cham, pp. 259–277.

[95] Fedyukovich, G. and Rümmer, P. [2021]. Competition report: CHC-COMP-21, *in* H. Hojjat and B. Kafle (eds), *Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021*, Vol. 344 of *EPTCS*, pp. 91–108.

[96] Ferrara, A. L., Madhusudan, P., Nguyen, T. L. and Parlato, G. [2014]. VAC - verifier of administrative role-based access control policies, *in* A. Biere and R. Bloem (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 184–191.

[97] Floyd, R. W. [1967]. Assigning meanings to programs, *Mathematical Aspects of Computer Science* **19**: 19–32.

[98] 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*, Springer International Publishing, Cham, pp. 58–76.

[99] Fuchs, A., Goel, A., Grundy, J., Krstic, S. and Tinelli, C. [2012]. Ground interpolation for the theory of equality, *Logical Methods in Computer Science* **8**(1).

[100] Graf, S. and Saidi, H. [1997]. Construction of abstract state graphs with PVS, *in* O. Grumberg (ed.), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 72–83.

[101] Grebenshchikov, S., Lopes, N. P., Popeea, C. and Rybalchenko, A. [2012]. Synthesizing software verifiers from proof rules, *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, Association for Computing Machinery, New York, NY, USA, pp. 405–416.

[102] Griggio, A. [2012]. A practical approach to satisability modulo linear integer arithmetic, *J. Satisf. Boolean Model. Comput.* **8**: 1–27.

[103] Griggio, A., Le, T. T. H. and Sebastiani, R. [2012]. Efficient interpolant generation in satisfiability modulo linear integer arithmetic, *Logical Methods in Computer Science* **8**(3).

[104] Griggio, A. and Roveri, M. [2016]. Comparing different variants of the IC3 algorithm for hardware model checking, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* **35**(6): 1026–1039.

[105] Gurfinkel, A. and Bjørner, N. [2019]. The science, art, and magic of constrained Horn clauses, *2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pp. 6–10.

[106] Gurfinkel, A. and Ivrii, A. [2017]. K-induction without unrolling, *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 148–155.

[107] Gurfinkel, A., Kahsai, T., Komuravelli, A. and Navas, J. A. [2015]. The SeaHorn verification framework, *in* D. Kroening and C. S. Păsăreanu (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 343–361.

[108] Gurfinkel, A., Rollini, S. F. and Sharygina, N. [2013]. Interpolation properties and SAT-based model checking, *in* D. Van Hung and M. Ogawa (eds), *ATVA 2013*, Springer, Cham, pp. 255–271.

[109] Gurfinkel, A. and Vizel, Y. [2014]. DRUPing for interpolants, *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD ’14, FMCAD Inc, Austin, Texas, pp. 99–106.

[110] Heizmann, M., Hoenicke, J. and Podelski, A. [2013]. Software model checking for people who love automata, *in* N. Sharygina and H. Veith (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 36–52.

[111] Henzinger, T. A., Jhala, R., Majumdar, R. and Sutre, G. [2002]. Lazy abstraction, *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’02, Association for Computing Machinery, New York, NY, USA, pp. 58–70.

[112] Hoare, C. A. R. [1969]. An axiomatic basis for computer programming, *Commun. ACM* **12**(10): 576–580.
URL: *https://doi.org/10.1145/363235.363259*

[113] Hoder, K. and Bjørner, N. [2012]. Generalized property directed reachability, *in* A. Cimatti and R. Sebastiani (eds), *Theory and Applications of Satisfiability Testing – SAT 2012*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 157–171.

[114] Hoenicke, J. and Schindler, T. [2018]. Efficient interpolation for the theory of arrays, *in* D. Galmiche, S. Schulz and R. Sebastiani (eds), *Automated Reasoning*, Springer International Publishing, Cham, pp. 549–565.

[115] Hojjat, H., Iosif, R., Konečný, F., Kuncak, V. and Rümmer, P. [2012]. Accelerating interpolants, *in* S. Chakraborty and M. Mukund (eds), *Automated Technology for Verification and Analysis*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 187–202.

[116] Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V. and Rümmer, P. [2012]. A verification toolkit for numerical transition systems, *in* D. Giannakopoulou and D. Méry (eds), *FM 2012: Formal Methods*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 247–251.

[117] Hojjat, H. and Rümmer, P. [2018]. The ELDARICA Horn solver, *FMCAD*, IEEE, pp. 158–164.

[118] Hojjat, H., Rümmer, P., Subotic, P. and Yi, W. [2014]. Horn clauses for communicating timed systems, *Electronic Proceedings in Theoretical Computer Science* **169**: 39–52.

[119] Horbach, M., Voigt, M. and Weidenbach, C. [2017]. The universal fragment of Presburger arithmetic with unary uninterpreted predicates is undecidable, *CoRR* **abs/1703.01212**.
URL: *http://arxiv.org/abs/1703.01212*

[120] Huang, G. [1995]. Constructing Craig interpolation formulas, *in* D.-Z. Du and M. Li (eds), *Computing and Combinatorics*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 181–190.

[121] Huberman, B. A., Lukose, R. M. and Hogg, T. [1997]. An economics approach to hard computational problems, *Science* **275**(5296): 51–54.

[122] 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. Le Berre (eds), *Theory and Applications of Satisfiability Testing – SAT 2016*, Springer International Publishing, Cham, pp. 547–553.

[123] Hyvärinen, A. E. J., Marescotti, M. and Sharygina, N. [2015]. Search-space partitioning for parallelizing SMT solvers, *in* M. Heule and S. Weaver (eds), *Theory and Applications of Satisfiability Testing – SAT 2015*, Springer International Publishing, Cham, pp. 369–386.

[124] Jančík, P., Alt, L., Fedyukovich, G., Hyvärinen, A. E. J., Kofroň, J. and Sharygina, N. [2016]. PVAIR: Partial variable assignment interpolator, *in* P. Stevens and A. Wasowski (eds), *FASE 2016*, Springer, Heidelberg.

[125] Jančík, P., Kofroň, J., Rollini, S. F. and Sharygina, N. [2014]. On interpolants and variable assignments, *FMCAD 2014*, IEEE, pp. 123–130.

[126] Jhala, R. and Majumdar, R. [2009]. Software model checking, *ACM Comput. Surv.* **41**(4).

[127] Jhala, R. and McMillan, K. L. [2005]. Interpolant-based transition relation approximation, *in* K. Etessami and S. K. Rajamani (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 39–51.

[128] Jhala, R. and McMillan, K. L. [2006]. A practical and complete approach to predicate refinement, *in* H. Hermanns and J. Palsberg (eds), *Tools and Algorithms for the Construction and Analysis of Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 459–473.

[129] Jovanović, D. and Dutertre, B. [2016]. Property-directed k-induction, *2016 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 85–92.

[130] Kafle, B. and Gallagher, J. P. [2015]. Tree automata-based refinement with application to Horn clause verification, *in* D. D'Souza, A. Lal and K. G. Larsen (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 209–226.

[131] Kafle, B., Gallagher, J. P. and Ganty, P. [2016]. Solving non-linear Horn clauses using a linear Horn clause solver, *Electronic Proceedings in Theoretical Computer Science* **219**: 33–48.

[132] Kahsai, T., Rümmer, P., Sanchez, H. and Schäf, M. [2016]. Jayhorn: A framework for verifying Java programs, *in* S. Chaudhuri and A. Farzan (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 352–358.

[133] Kahsai, T. and Tinelli, C. [2011]. PKind: A parallel k-induction based model checker, *Electronic Proceedings in Theoretical Computer Science* **72**: 55–62.

[134] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S. [2009]. SeL4: Formal verification of an OS kernel, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, Association for Computing Machinery, New York, NY, USA, pp. 207–220.
URL: *https://doi.org/10.1145/1629575.1629596*

[135] Komuravelli, A., Bjørner, N., Gurfinkel, A. and McMillan, K. L. [2015]. Compositional verification of procedural programs using Horn clauses over integers and arrays, *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 89–96.

[136] Komuravelli, A., Gurfinkel, A. and Chaki, S. [2014]. SMT-based model checking for recursive programs, *in* A. Biere and R. Bloem (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 17–34.

[137] Komuravelli, A., Gurfinkel, A. and Chaki, S. [2016]. SMT-based model checking for recursive programs, *Formal Methods in System Design* **48**(3): 175–205.

[138] Korovin, K., Tsiskaridze, N. and Voronkov, A. [2009]. Conflict resolution, *in* I. P. Gent (ed.), *CP 2009*, Springer, Heidelberg, pp. 509–523.

[139] Krajíček, J. [1997]. Interpolation theorems, lower bounds for proof systems, and independence results for bounded arithmetic, *The Journal of Symbolic Logic* **62**(2): 457–486.

[140] Kroening, D., Lewis, M. and Weissenbacher, G. [2015]. Under-approximating loops in C programs for fast counterexample detection, *Formal Methods in System Design* **47**(1): 75–92.

[141] Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A. and Wintersteiger, C. M. [2013]. Loop summarization using state and transition invariants, *Formal Methods in System Design* **42**(3): 221–261.

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

[143] Kroening, D. and Weissenbacher, G. [2011]. Interpolation-based software verification with Wolverine, *in* G. Gopalakrishnan and S. Qadeer (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 573–578.

[144] Lange, T., Prinz, F., Neuhäußer, M. R., Noll, T. and Katoen, J.-P. [2018]. Improving generalization in software IC3, *in* M. d. M. Gallardo and P. Merino (eds), *Model Checking Software*, Springer International Publishing, Cham, pp. 85–102.

[145] Leino, K. R. M. [2010]. Dafny: An automatic program verifier for functional correctness, *in* E. M. Clarke and A. Voronkov (eds), *Logic for Programming, Artificial Intelligence, and Reasoning*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 348–370.

[146] Leroux, J., Rümmer, P. and Subotić, P. [2016]. Guiding Craig interpolation with domain-specific abstractions, *Acta Informatica* **53**(4): 387–424.

[147] Leroy, X. [2009]. Formal verification of a realistic compiler, *Commun. ACM* **52**(7): 107–115.
URL: *https://doi.org/10.1145/1538788.1538814*

[148] Leroy, X. [2022]. The CompCert C verified compiler, `https://compcert.org/man/`. Accessed: January 2022.

[149] Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A. and Barrett, C. [2021]. Pono: A flexible and extensible SMT-based model checker, *in* A. Silva and K. R. M. Leino (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 461–474.

[150] Marescotti, M., Gurfinkel, A., Hyvärinen, A. E. J. and Sharygina, N. [2017]. Designing parallel PDR, *2017 Formal Methods in Computer Aided Design (FMCAD)*, pp. 156–163.

[151] Marescotti, M., Hyvärinen, A. E. J. and Sharygina, N. [2016]. Clause sharing and partitioning for cloud-based SMT solving, *in* C. Artho, A. Legay and D. Peled (eds), *Automated Technology for Verification and Analysis*, Springer International Publishing, Cham, pp. 428–443.

[152] Marescotti, M., Hyvärinen, A. E. J. and Sharygina, N. [2018]. SMTS: distributed, visualized constraint solving, *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Awassa, Ethiopia, 16-21 November 2018*.

[153] 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*, Springer International Publishing, Cham, pp. 178–194.

[154] Matsushita, Y., Tsukada, T. and Kobayashi, N. [2021]. RustHorn: CHC-based verification for Rust programs, *ACM Trans. Program. Lang. Syst.* **43**(4).

[155] McMillan, K. L. [2003]. Interpolation and SAT-based model checking, *in* W. A. Hunt and F. Somenzi (eds), *Computer Aided Verification*, Springer, Berlin Heidelberg, pp. 1–13.

[156] McMillan, K. L. [2005]. An interpolating theorem prover, *Theoretical Computer Science* **345**(1): 101–121.

[157] McMillan, K. L. [2006]. Lazy abstraction with interpolants, *in* T. Ball and R. B. Jones (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 123–136.

[158] McMillan, K. L. [2010]. Lazy annotation for program testing and verification, *Computer Aided Verification (CAV' 10)*, Vol. 6174 of *LNCS*, pp. 104–118.

[159] McMillan, K. L. [2014]. Lazy annotation revisited, *Proc. CAV 2014*, Vol. 8559 of *LNCS*, Springer, pp. 243–259.

[160] McMillan, K. L. and Rybalchenko, A. [2013]. Solving constrained Horn clauses using interpolation, *Technical report*, Microsoft Research.

[161] Möller, B. [2011]. Binary exponentiation, *in* H. C. A. van Tilborg and S. Jajodia (eds), *Encyclopedia of Cryptography and Security*, Springer US, Boston, MA, pp. 84–86.

[162] Palikareva, H. and Cadar, C. [2013]. Multi-solver support in symbolic execution, *in* N. Sharygina and H. Veith (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 53–68.

[163] Piterman, N. and Pnueli, A. [2018]. *Temporal Logic and Fair Discrete Systems*, Springer International Publishing, Cham, pp. 27–73.

[164] Podelski, A. and Rybalchenko, A. [2004]. Transition invariants, *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, 2004.*, pp. 32–41.

[165] Podelski, A. and Rybalchenko, A. [2005]. Transition predicate abstraction and fair termination, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, Association for Computing Machinery, New York, NY, USA, pp. 132–144.

[166] Pudlák, P. [1997]. Lower bounds for resolution and cutting plane proofs and monotone computations, *Journal of Symbolic Logic* **62**(3): 981–998.

[167] Queille, J. P. and Sifakis, J. [1982]. Specification and verification of concurrent systems in CESAR, *in* M. Dezani-Ciancaglini and U. Montanari (eds), *International Symposium on Programming*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 337–351.

[168] Rakadjiev, E., Shimosawa, T., Mine, H. and Oshima, S. [2015]. Parallel SMT solving and concurrent symbolic execution, *2015 IEEE Trustcom/BigDataSE/ISPA*, Vol. 3, pp. 17–26.

[169] Rebola-Pardo, A. and Weissenbacher, G. [2020]. RAT elimination, *in* E. Albert and L. Kovacs (eds), *LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Vol. 73 of *EPiC Series in Computing*, EasyChair, pp. 423–448.

[170] Rollini, S. F. [2014]. *Craig Interpolation and Proof Manipulation*, PhD thesis, Università della Svizzera italiana.

[171] Rollini, S. F., Alt, L., Fedyukovich, G., Hyvärinen, A. E. J. and Sharygina, N. [2013]. PeRIPLO: A framework for producing effective interpolants in SAT-based software verification, *in* K. McMillan, A. Middeldorp and A. Voronkov (eds), *LPAR 2013*, Vol. 8312 of *LNCS*, Springer, Heidelberg, pp. 683–693.

[172] Rollini, S. F., Bruttomesso, R., Sharygina, N. and Tsitovich, A. [2014]. Resolution proof transformation for compression and interpolation, *Formal Methods in System Design* **45**(1): 1–41.

[173] Rümmer, P. [2008]. A constraint sequent calculus for first-order logic with linear integer arithmetic, *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Vol. 5330 of *LNCS*, Springer, pp. 274–289.

[174] Rümmer, P. [2020]. Competition report: CHC-COMP-20, *Electronic Proceedings in Theoretical Computer Science* **320**: 197–219.

[175] Rümmer, P., Hojjat, H. and Kuncak, V. [2013]. Disjunctive interpolants for Horn-clause verification, *in* N. Sharygina and H. Veith (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 347–363.

[176] Rümmer, P., Hojjat, H. and Kuncak, V. [2015]. On recursion-free Horn clauses and Craig interpolation, *Formal Methods In System Design* **47**(1): 1–25.

[177] Rümmer, P. and Subotic, P. [2013]. Exploring interpolants, *Proc. FMCAD 2013*, IEEE, pp. 69–76.

[178] Rybalchenko, A. and Sofronie-Stokkermans, V. [2007]. Constraint solving for interpolation, *in* B. Cook and A. Podelski (eds), *Verification, Model Checking, and Abstract Interpretation*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 346–362.

[179] Schindler, T. and Jovanović, D. [2018]. Selfless interpolation for infinite-state model checking, *in* I. Dillig and J. Palsberg (eds), *VMCAI 2018*, Springer, Cham, pp. 495–515.

[180] Scholl, C., Pigorsch, F., Disch, S. and Althaus, E. [2014]. Simple interpolants for linear arithmetic, *DATE 2014*, IEEE, pp. 1–6.

[181] Schrijver, A. [1998]. *Theory of Linear and Integer Programming*, John Wiley & Sons, Inc., New York.

[182] Sery, O., Fedyukovich, G. and Sharygina, N. [2012a]. Incremental upgrade checking by means of interpolation-based function summaries, *2012 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 114–121.

[183] Sery, O., Fedyukovich, G. and Sharygina, N. [2012b]. Interpolation-based function summaries in bounded model checking, *Proc. HVC 2011*, Vol. 7261 of *LNCS*, Springer, pp. 160–175.

[184] Seufert, T. and Scholl, C. [2018]. Combining PDR and reverse PDR for hardware model checking, *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 49–54.

[185] Sharma, R., Dillig, I., Dillig, T. and Aiken, A. [2011]. Simplifying loop invariant generation using splitter predicates, *in* G. Gopalakrishnan and S. Qadeer (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 703–719.

[186] Sheeran, M., Singh, S. and Stålmarck, G. [2000]. Checking safety properties using induction and a SAT-solver, *in* W. A. Hunt and S. D. Johnson (eds), *Formal Methods in Computer-Aided Design*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 127–144.

[187] Tan, B., Mariano, B., Lahiri, S. K., Dillig, I. and Feng, Y. [2022]. SolType: Refinement types for arithmetic overflow in Solidity, *Proc. ACM Program. Lang.* **6**(POPL).

[188] Trakhtenbrot, B. [1984]. A survey of Russian approaches to perebor (brute-force searches) algorithms, *Annals of the History of Computing* **6**(4): 384–400.

[189] Unno, H., Terauchi, T. and Koskinen, E. [2021]. Constraint-based relational verification, *in* A. Silva and K. R. M. Leino (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 742–766.

[190] Vediramana Krishnan, H. G., Chen, Y., Shoham, S. and Gurfinkel, A. [2020]. Global guidance for local generalization in model checking, *in* S. K. Lahiri and C. Wang (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 101–125.

[191] Vediramana Krishnan, H. G., Fedyukovich, G. and Gurfinkel, A. [2020]. Word level property directed reachability, *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9.

[192] Vediramana Krishnan, H. G., Shoham, S. and Gurfinkel, A. [2022]. Solving constrained Horn clauses modulo algebraic data types and recursive functions, *Proc. ACM Program. Lang.* **6**(POPL).

[193] Vediramana Krishnan, H. G., Vizel, Y., Ganesh, V. and Gurfinkel, A. [2019]. Interpolating strong induction, *in* I. Dillig and S. Tasiran (eds), *Computer Aided Verification*, Springer International Publishing, Cham, pp. 367–385.

[194] Vizel, Y. and Grumberg, O. [2009]. Interpolation-sequence based model checking, *Proc. FMCAD 2014*, IEEE, pp. 1–8.

[195] Wang, W. and Jiao, L. [2016]. Trace Abstraction Refinement for Solving Horn Clauses, *The Computer Journal* **59**(8): 1236–1251.

[196] Weber, T., Conchon, S., Déharbe, D., Heizmann, M., Niemetz, A. and Reger, G. [2019]. The SMT competition 2015–2018, *Journal on Satisfiability, Boolean Modeling and Computation* **11**: 221–259.

[197] Weissenbacher, G. [2012]. Interpolant strength revisited, *Proc. SAT 2012*, Vol. 7317 of *LNCS*, Springer, pp. 312–326.

[198] Wieringa, S., Niemenmaa, M. and Heljanko, K. [2009]. Tarmo: A framework for parallelized bounded model checking, *Electronic Proceedings in Theoretical Computer Science* **14**: 62–76.

[199] Wintersteiger, C. M., Hamadi, Y. and de Moura, L. [2009]. A concurrent portfolio approach to SMT solving, *in* A. Bouajjani and O. Maler (eds), *Computer Aided Verification*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 715–720.

[200] Yorsh, G. and Musuvathi, M. [2005]. A combination method for generating interpolants, *in* R. Nieuwenhuis (ed.), *Automated Deduction – CADE-20*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 353–368.