Parallelization and modelling techniques for scalable SMT-based verification

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

presented by
Matteo Marescotti

under the supervision of
Natasha Sharygina

December 2020

Dissertation Committee

Walter Binder	Università della Svizzera Italiana, Lugano, Switzerland
Nikolaj Bjørner	Microsoft Research, Redmond, USA
Evanthia Papadopoulou	Università della Svizzera Italiana, Lugano, Switzerland
Roberto Sebastiani	Università di Trento, Italy

Dissertation accepted on 1 December 2020

Research Advisor	PhD Program Director
Natasha Sharygina	**The PhD program Director *pro tempore***

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

Matteo Marescotti
Lugano, 1 December 2020

Abstract

Software systems are increasingly involved in our daily life tasks, making their failures potentially damaging both in economic and human terms. As a result, proving the correctness of software is widely thought to be one of the most central challenges for computer science. Model checking is an award-winning (Turing award, 2007) technique for formally and automatically verifying systems, often capable to guarantee the correctness of non-trivial and practical programs. However, model checking is undecidable in the general case. Even when properly restricted to the decidable fragment, the algorithms face strong scalability issues preventing them to converge to the solution.

The goal of this thesis is to address such scalability issues in two orthogonal ways. First, the design of parallel solving techniques that exploit the massive amount of computational power offered by distributed computing environments. Second, the design of a modelling technique for effective verification of the new emergent technology of smart contracts running on blockchain systems.

The parallel solving research efforts consider both bounded and induction-based unbounded model checking, respectively achieved by instantiating multi-agent solving for the T-DPLL and IC3 algorithms. Multi-agent techniques leverage on the diversification and cooperation among sequential solvers executed in parallel. Diversification is achieved with different parallel search heuristics, and cooperation is performed with the exchange of information both during solving time and via upfront agreements. The effectivenesses of various multi-agent settings are empirically evaluated using the purpose-built framework SMTS.

Verification approaches for emerging technologies tend to either reuse tools for existing languages that often result in inefficient and even unsound models, or to apply generic techniques which typically require human intervention to succeed. This thesis provides a new direct modelling approach using constrained Horn clauses for the emerging technology of smart contracts. Such approach allows automatic solving exploiting the proposed efforts in improving inductive solvers scalability, and guarantees soundness for the safety proofs. Additionally, the smart-contracts-specific task of measuring the amount of computation

needed to execute a transaction, i.e. gas consumption, is considered in this thesis proposing an algorithmic solution. The effectiveness of the modelling technique is empirically evaluated over smart contracts deployed in the ETHEREUM blockchain using an implementation inside the official compiler for the smart contracts language Solidity that relies on the IC3 algorithm.

Acknowledgements

During my Ph.D. I have had the unique opportunity to meet and collaborate with brilliant people that inspired me to keep improving both professionally and personally. This thesis exists thanks to such interactions, and I am grateful for all of them. Such a fantastic journey started thanks to my advisor, Prof. Natasha Sharygina. I am grateful for her trust in me as a candidate, and for her generous and bold guidance that was essential throughout the years.

I thank the committee of this thesis, Prof. Evanthia Papadopoulou, Prof. Walter Binder, Dr. Nikolaj Bjørner, and Prof. Roberto Sebastiani for their efforts to thoroughly review this thesis and provide valuable comments. In particular, I thank Dr. Nikolaj Bjørner and Prof. Roberto Sebastiani for the enlightening discussions that sparked my curiosity and motivation. I thank Dr. Antti Hyvärinen for sharing with me bright insights and his vast experience across several topics. I thank Prof. Arie Gurfinkel for giving me the fantastic opportunity of visiting him in Waterloo, for his patience, and for sharing his sharp engineering skills.

Finally, I thank all my colleagues for being supportive and inclusive, I am grateful for the wonderful time spent in Lugano. I thank USI and the faculty staff for their quick and helpful assistance, and the Swiss National Science Foundation for the financial support received (projects 185031, 166288 and 153402).

Contents

Chapter 1

Introduction

Computer systems are a central piece of modern society: our daily lives involve plenty of tasks entrusted to software executing on various hardware systems. Crucial issues like people safety and proper enforcement of legal and monetary operations highly rely on correctly working software and hardware. Furthermore, industry safety, security, reliability, cost-efficiency and environmental friendliness critically depend on correctly working systems. Such relevance of computer systems is continuously increasing and makes the consequences of faults being potentially catastrophic.

The most robust way to prevent such unfortunate eventualities is to *formally verify* the systems, that is to check that systems behave correctly in any possible circumstances during the execution, therefore *proving* correctness. However, real-world systems present vast functionalities that prevent such check to be performed trivially. For these reasons, proving the correctness of computer systems is widely thought to be one of the most central challenges for computer science. Formal methods address the verification problem proving correctness in a mathematical sense, showing with an inferential argument that the system behaviours logically guarantee correctness. Formal verification is *automated* when no human intervention occurs in the complex process of devising a proof. Automation excludes human errors from altering the overall result at the price of increasing even more the complexity of the verification task, that needs to account for the reasoning independently and thoroughly. Therefore, automation opens several challenges in devising efficient reasoning methods.

This thesis focuses on improving the reasoning capabilities of automated verification techniques for software systems, contributing in two orthogonal ways to this highly challenging task. First, it is achieved by exploiting the high computational power offered by distributed computing systems. Second, it is suggested

to address the problem by designing modelling techniques necessary to correctly verify the new emerging technology of smart contracts running on blockchain systems. Additionally, this thesis studies the problem of finding the worst-case gas consumption of smart contracts transactions, presenting a solution specific for this emerging context. The introduction of this thesis provides an overview over the state of the affairs of automated formal verification, showing in section 1.1 the various different directions that aim to cope with the problem. Then, section 1.2 states open challenges, and finally section 1.3 presents an overview of the proposed solutions.

1.1 Automated Formal Verification

The aim of *hardware* and *software verification* is to check the correctness of a given input system by detecting the presence or the absence of behaviours that violate the specification. Verification can be addressed dynamically with *testing*, or statically with *formal verification*. Testing is performed by executing the system either entirely or partially on predefined test inputs which values are based on the expected production settings. All testing algorithms suffer from *needle-in-haystack* problem: there is no general way to detect and test the set of inputs that would trigger undesired behaviours. For this reason testing is considered an incomplete technique: it can produce only proofs of incorrectness. The only way to prove the absence of bad behaviours is by testing the correctness of all possible inputs, i.e. the proof by exhaustion, that unfortunately is only feasible in very restricted circumstances, leaving testing incomplete over real world problems. This thesis focuses on complete techniques, thus complementing testing with exhaustive verification solutions.

The aim of formal verification is to *prove* that the input system complies the given specifications in a mathematical way. The proof ensures that no possible violation of the specification can happen anytime during any possible execution. Despite the fact that the general approach is undecidable, research is moving forward to find more and more restricted scopes in which it is decidable and works thoroughly. Formal verification approaches can be distinguished between *interactive* and *automatic* verification. Interactive verification is a user driven procedure involving the discharging of mathematical statements using theorem provers. Interactive verification approaches require human interaction, resulting in two important side effects. First, they require experienced manual effort that increases the time and cost of verification processes. Second, human intervention could accidentally introduce bugs also in the verification process, making the

outcome less reliable. For these reasons, interactive techniques are of limited use by developers and audit communities.

Automated formal verification techniques perform a fully automatic and static analysis of the system, that is without human intervention and without executing the system, respectively. A successful automated formal verification approach is *model checking* [CE81, QS82, CGP01]. Model checking constructs a *model* that formally represents the input system as a transition system over internal states. The model expresses both internal initial states and state transitions with respect of external inputs. The *properties* are built from the requirements of the system and are formalized using temporal logic [CGP01]. An important class of temporal logic is *safety*. Safety properties describe bad behaviours as a set of finite-length executions that the system must never produce in order to be safe. Safety properties are particularly important in verification since they can be expressed as a reachability problem, thus making the check relatively efficient. Model checking algorithms perform the exhaustive search of property violations in every state of the model in a fully automatic way. Therefore, a positive result with respect of safety ensures that the properties hold in any circumstances during execution. Conversely, a negative result provides a *counterexample*, that is, the state where the violation is found together with the reachability *trace* of states traversed during the search from the initial states. The counterexample is a convenient way for developers to understand the problem and design a solution.

An important downside of model checking is the combinatorial blow up in the number of states of the system, commonly known as *state space explosion*. Although hardware systems are finite state systems, software does not have such restriction in general, therefore state space explosion on software model checking could cause the model to become infinitely large. This issue must be addressed in most real-world problems which would be not tractable otherwise.

Symbolic model checking [CGP01] is an active area of research that faces state space explosion by modelling the state and the transition relation of the system respectively with a set of variables and logical formulas over such variables. A symbolic value expresses any possible concrete value which can be assigned to a given variable in any possible execution. The model of the transition relation models how variables are manipulated during execution. The property is also modelled as a formula describing all unsafe executions. Thus, the conjunction of the model and the property is a formula which satisfiability directly depends on whether the system can produce property violations. In this way, symbolic model checking reduces the problem of exhaustively checking reachability of violations to determining the satisfiability of logical formulas.

Model checking is *bounded* [BCCZ99] (BMC) when the verification algorithm

preprocesses the source code by unwinding the program loops up to a fixed bound. BMC checks programs exhaustively and automatically over the states reachable in up to a certain number of loop iterations. As a result, bounded safety proofs are *partial* because they consider a possible subset of the reachable states. However, BMC is a successful approach for finding bugs in real world programs, especially if bugs can manifest in a limited number of loop iterations. Model checking is *unbounded* when the produced safety proofs are not bounded by the number of loop iterations, i.e. they consider all possible reachable states of the system. Unbounded model checking techniques aim to find an *invariant* that proves safety, i.e. a formula that is valid in every reachable state and implies the property.

Both bounded and unbounded model checkers build propositional or first-order logic models, and rely on satisfiability oracles for proving or refuting safety. For this purpose, model checkers use automatic reasoning engines such as SAT and SMT (*Satisfiability Modulo Theories*) [DNS05] solvers as oracles. SAT solvers are tools designed to find whether a boolean formula given as input in conjunctive normal form (CNF) is satisfiable, i.e. there exists an assignment of each boolean variable that makes the formula true. The SMT problem consists of determining whether a propositional formula is satisfiable, given that some of the propositional atoms have an interpretation in first-order logic. Both SAT and SMT solvers are highly engineered tools having strong automatic reasoning power that can address this challenge.

In this way, coping with the high complexity of verification is entirely entrusted to the algorithms underlying the solving task, posing several challenges in devising efficient model checking techniques.

1.2 Verification challenges addressed in the thesis

In the last decades symbolic model checking has increasingly gained attention from both academia and industry, resulting in major efforts for improving both modelling techniques and solving capabilities. However, despite the promising positive trend, often symbolic model checking still fails to face real-word verification tasks. The causes that prevent scaling to real-world problems can be found both in the solving task and the modelling. This thesis studies solutions for improvement on both sides.

The solving task relies on automated reasoning tools such SAT or SMT solvers. This step is the bottleneck in symbolic model checking techniques, as the underlying decision problems are intrinsically computationally hard. As a result,

algorithms used for solving are conceptually challenging, often complicated to implement and limited in reasoning capabilities. *Abstraction* is a central technique used by automated reasoning to face the state-space explosion problem and address verification of infinite-state systems. Abstraction involves removing details expected to not be needed for proving the desired property. In predicate abstraction [GS97] concrete states that satisfy a given predicate are clustered in *abstract states*. Infinite-state systems can be represented by a finite-state abstracted system, guaranteed to over-approximate the reachable states of the original system, thus preventing state-space explosion and maintaining soundness. The crucial challenge for abstraction is to devise heuristics that correctly decide which details are important and which can be abstracted out. The amount of information removed defines the *level of abstraction*. The problem of finding the best level of abstraction for a given verification task is still an active research area. A considerable improvement on this front is the counterexample-guided abstraction refinement (CEGAR) [CGJ+00] framework. The idea is to iteratively refine the abstraction by adjusting the level based on the counterexample returned by the check. Similarly, BMC implements abstraction techniques for simplifying the program into an easier one that behaves in the same way with respect to the property being checked. *Craig interpolation* [Cra57] is a widely used approach for constructing a mathematical formula that over-approximates (i.e. is implied by) the original formula representing the input program. Interpolation is also used intensively in unbounded model checking to devise an inductive invariant.

Such techniques designed to deal with the state-space explosion problem to improve scalability are heuristics-based and sequential, therefore intrinsically limited by the high complexity. Being the problem undecidable entails that an heuristic performing the best on every kind of verification task does not exist. Rather, some heuristics perform well on some tasks while being inefficient on others. For this reason, improving a particular heuristic is often very complicated and the result might provide limited benefits. This is the ideal picture where parallel computing has potentials for being highly beneficial: distributed computing clusters offer massive amounts of computing power and sequential heuristics are intrinsically imprecise and often make mistakes. The aim of parallel techniques is to use concurrent computation to avoid such imprecision to waste precious wall-clock time.

Multi-agent solving is the parallel execution of a collection of solvers, the agents, that cooperate in some ways for solving the same problem. Possible cooperation techniques among agents are the exchange of learnt information performed during solving, or the agreement performed upfront to divide the solving task in a way that the final solution is the result of the independent collabora-

tion. Diversity in the agents search heuristics is a key property to maximize the search-space coverage and thus the likelihood to find a solution. From this line of work several new challenges arise. First, how to properly combine the cooperation and diversification techniques to improve scalability is highly nontrivial. Second, the information exchanged by the solvers need heuristics to filter useless or redundant data that might burden the individual sequential algorithms, and must be proved to not break soundness. Finally, the parallel system might easily become hard to control and debug, therefore needing visual tools to understand and interact with the multi-agent system. The multi-agent techniques proposed in this thesis focus on the challenges (as described above) arising for parallelizing both bounded and unbounded model checking.

The other research problem addressed in this thesis is the issue of appropriate formal modelling. The modelling task presents a trade-off between ease of modelling and effectiveness of adapting to the solving task. It is of crucial importance for choosing the modelling technique to consider both the input system and which and how automated reasoning tools are in charge of solving the models. Modelling techniques failing this goal might cause strong inefficiency or even unsound results on the solving side. Circumstances under which such trade-off arises are, for instance, when dealing with new systems: reusing existing modelling tools often eases the verification design with the risk of worsening the verification performances.

This thesis addresses this challenge by looking at the emergent field of smart contracts. Smart contracts is a technique to manage and enforce contract transactions without relying on trusted parties but instead exploiting the blockchain technology to distribute trust among peers. The safety of smart contracts is increasingly important, making formal methods a urgent need in this area. In fact, in the past years millions of US Dollars were lost due to bugs [the20, par17], while currently the smart contracts deployed in the widely used ETHEREUM platform control billions of dollars of wealth. This need is even more pronounced because once deployed in the blockchain and made publicly available, the program code of smart contracts is immutable, complicating the approach used in more traditional software development of fixing errors with new releases.

The most popular languages for writing smart contracts are Turing-complete, and therefore deciding their correctness with respect to non-trivial semantic properties is undecidable. Current automated verification solutions exploit generic symbolic execution techniques used for traditional software verification. However, the reuse of such existing frameworks in order to profit from their stability and speed can introduce problems. In particular, existing verification languages do not support the specific semantics of smart contracts directly, and it is very

difficult to capture these semantics with the existing abstractions. This is often solved by allowing imprecise models for the verification. For example, the smart contract model checker ZEUS [KGDS18] uses the C-based LLVM bit-code [LA03] as an intermediate representation for Solidity contracts. However, this introduces behaviours not present in the original smart contract, for example arising from the C-memory model.

The obvious solution (as also presented in this thesis) to this problem is to develop smart contract-specific techniques. Examples of such projects include K-FRAMEWORK [RS10] and SECURIFY [TDDC+18]. K-FRAMEWORK is a special-purpose theorem prover which relies on user input, making its use time-consuming and difficult for non-experts. SECURIFY uses an incomplete method based on searching patterns focusing on data. However, in this approach some properties cannot be modelled as data patterns, but instead require a more precise analysis of the flow of program control. As a results, the system can produce spurious results that can confuse developers.

This overview over current smart contracts verification techniques motivates the need for a modelling technique that excludes imprecise results while being able to target automatic checking algorithms.

1.3 Contributions

This section presents the contributions of this thesis to face the challenges previously discussed. The contributions are grouped in three sections. Section 1.3.1 presents the contribution related to multi-agent SMT and BMC, Section 1.3.2 focuses on multi-agent unbounded model checking, finally Section 1.3.3 presents the solutions regarding smart contracts challenges.

1.3.1 Parallel SMT Solving

The SMT problem of determining the satisfiability of a first-order formula is in general undecidable. However, the SMT approach to first-order formula solving aims at being very practical by focusing on the needs of specific user communities. This focus on user communities allows the development of algorithms that are often highly efficient in the decidable fragments of first-order theories required by users' applications. The studies reported in this thesis focus on the *Davis–Putnam–Logemann–Loveland* algorithm incorporating reasoning about a theory T (T-DPLL) [Tin02, NOT06]. T-DPLL is the most successful and current state-of-the-art algorithm for SMT solving. T-DPLL solvers consist of a *Conflict-*

Driven Clause Learning (CDCL) SAT solver as the underlying search engine for the pure propositional boolean part, enhanced with theory-specific decision procedures for determining the consistency of theory atoms with respect to the theories. The specialized solvers, in case of unsatisfiability, return propositional clauses that explain the unsatisfiability and are tautologies in the first-order theory. In the formal verification domain, SMT solvers gained success quickly because they enabled effective reasoning of code behaviours without expressing every possible bit-level state as needed by SAT solvers. This extension makes formal verification scalable on many more real world problems.

Diversification in multi-agent SMT solving is provided by the *SMT portfolio* [HJS09], the concurrent and independent execution of several solvers provided with the same input formula. Portfolio takes advantage from the highly non-deterministic (or seed-based) heuristics that drive the search by executing several solvers concurrently. Cooperation among portfolio solvers is performed with *clause sharing*. Both SAT and SMT solvers learn a new clause on every conflict encountered during solving, both at the propositional and at the theory level. Learnt clauses are important because they represent information regarding previous decisions that were after discovered being wrong. Thus, the exchange of learnt clauses instructs solvers to not take such decisions, avoiding them to explore parts of the search-space where there is no solution. Finally, *search-space partitioning* of SMT instances [HJN10] involve the construction of a fixed number of partitions in such a way that they do not share any model, and whose disjunction is equisatisfiable to the given input formula. The partitions can be solved independently and each partition can be potentially partitioned again, implementing in this way *iterative partitioning*.

Chapter 3 focuses on how parallelism, and in particular distributed computing, can make T-DPLL-style solvers to scale to increasingly hard problems. When used in isolation, portfolio, clause sharing and partitioning suffer from scalability issues. The optimal tuning highly depends both on the right combination with other techniques and on the instance being solved. The challenges in this context center around the design of effective heuristics for making parallel SMT solving scalable, investigating ways to exploit each technique strengths for achieving effective performances. The central concept introduced in chapter 3 for allowing flexible tuning of multi-agent SMT solving is the algorithmic framework called *parallelization tree* [HMS15]. The framework allows combining the three parallelization approaches algorithm portfolios, partitioning, and clause sharing. The key idea of the framework is that both solving and partitioning the search space can be done with a portfolio while sharing clauses among the solvers.

1.3.2 Parallel Unbounded Model Checking

The goal of unbounded model checking techniques is to ensure that every finite-length execution of hardware and software systems respects the given properties. This task is equivalent to solving the reachability problem in a model represented as a transition system over system states. Given a set of initial states, a transition relation and a set of target states, the reachability problem consists of finding whether there exists a path from the initial states to the target states by taking any finite-length path of transition relation steps. In order to prove unbounded safety, the target states consist of all states violating the properties (error states). A proof that no such paths exist for any length involves finding an inductive invariant of the transition system that excludes the error states. The reachability problem is in general undecidable.

The IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness, a.k.a. PDR, Property Directed Reachability) [Bra11] algorithm is a recent specific technique to solve the reachability problem. It gained success quickly because of its efficiency of flexibility. IC3 proceeds by iteratively proving or disproving reachability of *proof obligations*, i.e. states that are guaranteed to reach an error state. The search proceeds backward from the error states, which constitute the first proof obligation. Then, IC3 iteratively selects a proof obligation and tries to prove its reachability from the initial states. The procedure computes the set of states leading in one transition step to the proof obligations, which becomes a new proof obligation to be selected later. When a proof obligation is proven not reachable, IC3 learns an IC3-lemma that by construction over-approximates the reachable states of the system. The iteration continues until the conjunction of all the learnt IC3-lemmas is a formula strong enough to block any other proof obligation inductively, making it a safe inductive invariant.

Chapter 4 studies and evaluates several combination of multi-agent techniques for IC3. Heuristics play a crucial role in IC3 both for deciding which path to take next and which proof obligations need priority, because of the underlying nature of the undecidability of the problem, exclude the existence of a general optimal heuristic. One of the goals of this thesis is to study how to combine the key principles of diversification and cooperation in multi-agent solving in order to achieve good performance and scalability when solving the reachability problem using IC3. Diversification is provided with a *IC3 portfolio*: the concurrent and independent execution on several IC3 heuristics on the same transition system. The IC3-lemmas are the results of reasoning efforts made by a specific heuristic, and can be gathered and used by all the solvers to soundly refine their reachability representation. The similarities with SMT suggest that the same principles for

parallelization can also be adapted to fit the IC3 context. An important improvement to IC3 portfolio is sharing IC3-lemmas among the parallel solving agents in order to achieve cooperation by exchanging information. Partitioning the solvers' search space can be done by having each solver focused on a subset of the proof obligations. Soundness is guaranteed when all proof obligations are considered by at least one IC3 execution.

1.3.3 Modelling Smart Contracts

Smart contracts are programs designed to manage and enforce contracts without relying on trusted parties, exploiting the blockchain technology to achieve distributed consensus among peers. The safety of smart contracts is increasingly important: in the past years millions of US Dollars were lost due to bugs [the20, par17], and currently the smart contracts deployed in the widely used ETHEREUM platform control increasing amounts of wealth in the order of billions of dollars. This issue is even more pronounced because once deployed in the blockchain, the source code of smart contracts is immutable, complicating the task of fixing errors with new releases. ETHEREUM [Eth18a] nowadays is the most popular smart contracts platform. Smart contracts in ETHEREUM are written in high-level languages such as Solidity [sol20] and Vyper [vyp20], that are compiled to the low-level EVM (ETHEREUM virtual machine) bytecode deployed in the blockchain.

Notably, in the ETHEREUM settings, the corresponding smart contract implementations have distinct features which set them apart from traditional software and introduce challenges on the modelling design. The most pronounced smart-contract-specific features are as follows: (1) the transactional nature of function calls – in case of error a function execution reverts the system state as if the function had not been called, (2) the re-entrancy feature – the control returning back to the original contract after a transactional call to a possible unknown external contract, and (3) gas consumption – contract transactions consumes a quantity of *gas* that depends on the amount of computation needed to perform the execution and on the storage usage. In addition to the listed above specifics, contract invariants, i.e. formulas that hold after any possible transactions, are of paramount importance for ensuring properties of smart contracts (e.g., double spending or the correctness of account balances).

Solidity and EVM are Turing-complete languages, therefore deciding contract correctness with respect to non-trivial properties is undecidable. Many highly optimized frameworks already exist for software verification, exploiting generic symbolic execution techniques to address undecidability. While such existing frameworks are often stable and efficient, adapting them to smart con-

tracts can introduce problems, as existing verification languages do not provide direct support for the smart contract-specific features. This semantic gap is often solved by allowing 'imprecise modeling'. For example, ZEUS [KGDS18] and SAFEVM [ACG+19] use verifiers for the C language that rely on the very different memory model of C, possibly causing imprecise results. An orthogonal approach is followed by K-FRAMEWORK [RS10] and SECURIFY [TDDC+18]. Although their modelling is specific for smart contracts, both approaches suffer from important downsides that limit precision and therefore wide adoption, as the former requires the interaction of highly experienced users and the latter relies on an incomplete method based on data patterns that limits modelling capabilities of e.g. the control flow. Chapter 6 considers the specific traits (1) and (2) from above to design a modelling technique that enables automatic verification of safety properties for smart contracts. The produced models reflect smart contracts semantics in an accurate way and allows automatic verification, in order to encourage users without deep expertise in formal methods to verify smart contracts. The modelling technique produces a first-order formula using constrained Horn clauses (CHC) [BGMR15], allowing modularity for the solving side. In fact, any solver supporting inductive reasoning, such IC3 and PD-KIND can perform the automatic check. The produced safe inductive invariants returned by the solver in case of safety is a contract invariant that ensures the developer about properties of contract's state. The produced interpretation of first-order relations in case on unsafe smart contracts is directly mapped to a list of transactions that the developer can use as a counterexample to reproduce and tackle the issue. Furthermore, the chapter evaluates the performance of the modelling technique over thousands on Solidity contracts deployed in the ETHEREUM blockchain using an implementation developed inside the official Solidity compiler [Eth18b], and compares with all the publicly available tools at the time of writing.

Another issue studies in this thesis is the problems related to estimating how the gas consumption of a transaction changes over the lifetime of a contract is a non-trivial task of extreme importance, since early-stage gas analysis can prevent contracts from becoming useless due to unforeseen increase in the necessary amount of gas needed. Chapter 7 focuses on the specific trait (3) to propose two bounded analysis techniques to calculate the worst-case gas consumption of contract functions.

1.4 Summary of Contributions

In summary, this thesis provides the following contributions.

- Chapter 3 focuses on the parallelization of the *T*-DPLL algorithm to achieve multi-agent cooperative SMT solving and shows empirical evidence of the beneficial effects. These results are published in [MHS16, HMS15].
- Chapter 4 focuses on the parallelization of the IC3 algorithm to achieve multi-agent cooperative inductive-based unbounded model checking and reports experimental results of the positive effects of several combination of techniques. Results for this contribution are published in [MGHS17].
- Chapter 5 focuses on the details of the framework called SMTS, developed to support the techniques presented in chapters 3 and 4, and designed to facilitate the implementation of multi-agent cooperative solvers over distributed computing environments based on existing sequential solvers. SMTS is used in chapters 3 and 4 to assess experimental evidence. Additionally, this chapter reports an external project that witness SMTS high flexibility in a related project. The results for this contribution are published in [MHS18].
- Chapter 6 focuses on an modelling technique for the emerging technology of smart contracts that considers its semantics in an accurate way, allows unbounded automatic reasoning for safety properties, and reports experimental results and comparison with related tools. These results are published in [MOA+20].
- Chapter 7 proposes a bounded technique to calculate the worst-case amount of gas needed to perform contract transactions. The results for this contribution are published in [MBH+18].

In addition to the list of publications above, two journal versions, each extending the contributions of chapters Chapters 3 and 4, are currently submitted to JSAT (`http://jsatjournal.org`) waiting response.

- Matteo Marescotti, Antti Hyvärinen and Natasha Sharygina. Designing Parallel SMT Solvers. Submitted in Feb. 2019.
- Matteo Marescotti, Antti Hyvärinen, Arie Gurfinkel and Natasha Sharygina. Designing Parallel IC3. Submitted in Oct. 2019.

Chapter 2

Background

This chapter introduces the background concepts and terminology necessary to support the contributions presented in this thesis.

2.1 Satisfiability

Given a set of *variables* and a set of *function symbols* $\mathscr{F}$ each associated with an *arity* $n \geq 0$, a *term* is either a variable or a function symbol with arity n applied to n terms. A function with arity 0 is a *constant*. Let $\mathscr{P}$ be a set of *predicate symbols*, each associated with an arity $n \geq 0$. The set $\mathscr{P}$ always contains the symbols $\top$ and $\bot$ of arity 0, and $=$ of arity 2. An application of a predicate symbol p with arity n on n terms is called an *atom*.

Given a finite set of atoms B, a *literal* is an atom or a negated atoms $x, \neg x$, $x \in B$, and a *clause* is a disjunction of literals. When negating literals the equation $\neg\neg x = x$ holds. A *propositional formula in conjunctive normal form* (CNF) is a conjunction of clauses. A clause is also referred to as a set of literals, and a CNF clause as a set of clauses. A *cube* is a conjunction of unit clauses referred to as a set of literals when this cannot be confused with a disjunction. A sequence of literals is denoted with $l_1 \dots l_n$, and, when the order plays no role, equate the sequence with the corresponding set $\{l_1, \dots, l_n\}$.

Definition 1 (Assignment) *An assignment $\sigma \subseteq \{x, \neg x \mid x \in B\}$ is a set of literals such that for no atom x, both $x \in \sigma$ and $\neg x \in \sigma$.*

An assignment is called *total* if for all atoms $x \in B$ either $x \in \sigma$ or $\neg x \in \sigma$. An atom x is *assigned* if either $x \in \sigma$ or $\neg x \in \sigma$. An assignment σ satisfies a clause c when $\sigma \cap c \neq \emptyset$. An assignment satisfies a CNF formula if all its clauses are satisfied.

Let U be a possibly infinite set of elements containing at least the truth values **true** and **false**. Given two elements a, b of U, $a \equiv b$ holds if and only if a and b are the same element. A *model* $\mathscr{M}$ assigns to each constant a unique element from U, to each function symbol of arity n a total function $U^n \to U$, to each predicate symbol of arity zero a truth value **true** or **false**, and to each predicate symbol of arity $n \geq 1$ a total function $U^n \to \{\textbf{true}, \textbf{false}\}$. An *interpretation* $\mathscr{A}$ is an extension of $\mathscr{M}$ to terms such that $t^{\mathscr{A}} \equiv t^{\mathscr{M}}$ when t is a constant or predicate symbol of arity zero, and $t^{\mathscr{A}} \equiv f^{\mathscr{M}}(t_1^{\mathscr{A}}, \ldots, t_n^{\mathscr{A}})$ if t is an application of the function symbol or predicate symbol f of arity $n \geq 1$ to terms $t_1, \ldots, t_n$. The interpretations of $\top^{\mathscr{A}}$ is **true** and the interpretation of $\bot^{\mathscr{A}}$ is **false**, and $t_1 =^{\mathscr{A}} t_2$ to be **true** if and only if both $t_1^{\mathscr{A}}$ and $t_2^{\mathscr{A}}$ denote the same element of U (i.e., $t_1^{\mathscr{A}} \equiv t_2^{\mathscr{A}}$).

A *theory* T is a non-empty set of models. A formula F is *T-satisfiable* if there exists a satisfying total assignment σ for F and an interpretation $\mathscr{A}$ that is an extension of a model $\mathscr{M} \in T$ such that for each $l \in \sigma$, $l^{\mathscr{A}} \equiv \textbf{true}$ if, for some atom x of F, l is of the form x; and $l^{\mathscr{A}} \equiv \textbf{false}$ if l is of the form $\neg x$. In particular, given a formula F and an assignment σ that is total (with respect to F), $\sigma \models_T F$ if σ is such an assignment. In addition $F' \models_p F$ if all assignments that satisfy F' also satisfy F propositionally. For a formula, clause, literal, or assignment ξ, $\mathrm{At}(\xi)$ is the set of atoms appearing in ξ.

In the theory of linear real arithmetic (LRA), the universe consists of real numbers, the function symbols are $*$ and $+$ of arity two restricted to expressing only linear terms, and the predicate symbol $\leq$; all three with their usual interpretations. The theory of uninterpreted functions with equality (EUF) places no restrictions on the interpretations of constants, functions, or predicates (apart from the inherent ones defined above for equality, $\top$, and $\bot$).

Visual representation of CNF Formulas. The *variable interaction graph* [Sin07] of a given CNF formula shows the Boolean structure by having each node representing a Boolean variable. Two variables a and b are connected with an edge (a, b) if there exists a clause c such that both a and b appear as literals in the clause. This representation is used for visualizing SAT instances up to thousands of variables in [Sin07]. In SMT, some of the Boolean variables have an interpretation with respect to the background theories. Such Boolean variables in general contain variables from the respective theory. Thus, Boolean variables that appear isolated with respect to the propositional CNF structure (i.e., do not appear in the same clause), might still be related at the theory level, sharing common theory variables. Therefore, variable interaction graphs for SMT instances of-

ten are poorly connected when considering only Boolean variables, and highly connected when considering theory-related literals with new edges.

2.2 Safety for Transition Systems

For a set of variables X, $X' = \{x' \mid x \in X\}$ is the set of fresh primed variables. The notation is extended to formulas, and write φ' for a formula obtained from φ by replacing all variables in φ with the corresponding primed variables. Furthermore $X^{[i]}$ is the set of variables obtained by adding i primes to each $x \in X$.

A program can be expressed as a transition system $\langle Init, Tr \rangle$ over variables X: the formula $Init(X)$ describes the program's initial states and the formula $Tr(X, X')$ describes the program's transition relation. Given a transition system over variables X, a *safety property* is the formula $\neg Bad(X)$. A set of states described by a formula $\varphi(X)$ is *safe* if $\varphi(X) \wedge Bad(X)$ is unsatisfiable. To keep the notation compact, when clear from the context this thesis refers to the formula $\varphi(X)$ and the transition relation $Tr(X, X')$ with simply φ and Tr respectively.

Definition 2 (Safety) *A transition system $\langle Init, Tr \rangle$ satisfies the safety property $\neg Bad$, i.e., the system is* safe *with respect to Bad, if all the states reachable from initial states Init with the transition relation Tr are safe.*

The transition system is safe up to k steps if its states reachable by i applications of the transition relation, for all $0 \leq i \leq k$, are safe. An instance of the safety problem is expressed as a triple $\mathscr{S} = \langle Init, Tr, Bad \rangle$. For simplicity, it is assumed that $Init \implies \neg Bad$. Otherwise, $\mathscr{S}$ is unsafe and the counterexample is a trivial model over X that satisfies $Init \wedge Bad$.

Definition 3 (Post Image) *Given a transition relation Tr and a set of states represented by F, the predicate $post^n_{Tr}(F)$ is the set of states reachable from any state in F after taking exactly n transitions of Tr. It is defined as follows:*

$$post^n_{Tr}(F) = \begin{cases} F & \text{if } n = 0, \\ \exists X' \cdot post^{n-1}_{Tr}(F)(X') \wedge Tr(X', X) & \text{if } n \geq 1. \end{cases}$$

$post^*_{Tr}(F)$ is the transitive closure of Tr:

$$post^*_{Tr}(F) = \bigvee_{n \geq 0} post^n_{Tr}(F)$$

The set of reachable states for a program $\mathscr{S}$ is $post^*_{Tr}(Init)$. The IC3 algorithm constructs an approximation of the reachable states by computing modularly overapproximations of states reachable by $\mathscr{S}$ in a certain number of steps. The algorithm represents these over-approximations with *IC3-lemmas*. An IC3-lemma is a formula over variables X describing reachability information learned by IC3 execution.

Definition 4 (Relatively Inductive and Invariant Lemmas) *Given initial states represented by Init, transition relation Tr and a set of IC3-lemmas F, an IC3-lemma* φ *is* inductive relative to F *if and only if*

$$Init \implies \varphi \qquad \text{and} \qquad \varphi \wedge F \wedge Tr \implies \varphi'$$

The formula φ *is an* inductive lemma *when it is inductive relative to true. A IC3-lemma* φ *is an* invariant lemma *if it is true in all the reachable states, i.e.,* $post^*_{Tr}(F) \implies \varphi$. *Every inductive IC3-lemma is invariant, but the converse is not true in general.*

An instance $\mathscr{S}$ is safe if there exists a safe inductive invariant Inv such that $Inv \implies \neg Bad$. $\mathscr{S}$ is unsafe if there exists an $n \in \mathbb{N}$ such that $post^n_{Tr}(Init) \wedge Bad$ is satisfiable. For an unsafe $\mathscr{S}$, a satisfying assignment for

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

is called a *feasible counterexample*. The satisfying assignment corresponds to a sequence of states where the first state satisfies *Init*, each consecutive pair of states satisfies *Tr*, and the final state satisfies *Bad*, and can, therefore, be considered as an evidence for a programming error.

2.3 The IcE/FiRE framework

The parallelization techniques proposed in this thesis are evaluated in many contexts. One of them is within a joint work published in [BHMS20] with other colleagues at USI. This section provides the necessary background and notation specific for this related context.

Given a safety problem $\mathscr{S} = \langle Init, Tr, Bad \rangle$, IcE/FiRE algorithms [BHMS20] work on an over-approximation $\mathscr{F}$ of the states of $\mathscr{S}$ reachable in n steps or less for some $n \geq 1$. The idea is to maintain the invariant that the predicate $\mathscr{F}$ does not intersect with *Bad*, while trying to prove that $\mathscr{F}$ is (k-)inductive. When $\mathscr{F}$

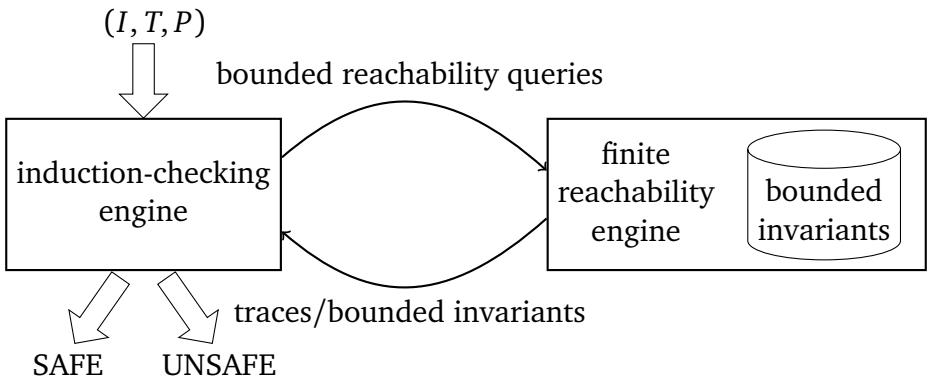


Figure 2.1. The IcE/FiRE framework for solving safety of transition systems

is represented symbolically as a set of formulas, individual elements of $\mathscr{F}$ can be checked for inductiveness relative to $\mathscr{F}$ instead of checking $\mathscr{F}$ as a whole. Successfully checked elements are collected in a new set $\mathscr{G}$ which represents an over-approximation of the states of $\mathscr{S}$ reachable in m steps or less, for $m > n$. When $\mathscr{G} = \mathscr{F}$, such an $\mathscr{F}$ (or $\mathscr{G}$) has the properties of a safe inductive invariant and therefore it is a solution for $\mathscr{S}$.

IcE/FiRE splits the reasoning about the safety into two separate components, as shown in fig. 2.1. The first, main, component is an *induction-checking engine* (IcE), also referred to shortly as induction engine. The goal of the induction engine is to decide the safety problem. It searches for a k-inductive strengthening of the property $\neg Bad$ 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 case of reachability, the reason is a trace of $n + 1$ states leading from an initial state to an s-state. In case of unreachability, the reason is an n-invariant blocking s.

The cooperation of these two engines is depicted on fig. 2.1. During the run, FiRE accumulates knowledge about the system in the form of bounded invariants. This knowledge helps it to answer the 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. The idea of separate components is introduced in [JD16] and allows to easily extend the framework to the parallel settings with information sharing. In addition, parallelization of IcE/FiRE covers not only PD-KIND [JD16], but also other algorithms, such as KIC3 [GI17].

Induction-Checking Engine

Given a safety problem for a transition system $\mathscr{S} = \langle Init, Tr, Bad \rangle$ the induction-checking engine (IcE) searches for k-inductive strengthening of $\neg Bad$. It maintains two distinct sets of state formulas: a *base* frame $\mathscr{F}$ and a *successor* frame $\mathscr{G}$. 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., Bad is not reachable in n steps or less. The base frame $\mathscr{F}$ serves both as a witness that Bad is not reachable, as well as a candidate for the inductive strengthening of $\neg Bad$. IcE maintains an invariant that on level n every element of $\mathscr{F}$ is an n-invariant. Moreover, $\neg Bad$ is always an element of $\mathscr{F}$. The successor frame $\mathscr{G}$ collects those elements of $\mathscr{F}$ that are $\mathscr{F}^k$-inductive for some fixed $k \leq n+1$. Since $\bigwedge \mathscr{F}$ is an n-invariant, this means that all elements of $\mathscr{G}$ are at least $(n+1)$-invariants. When all elements of the base frame are checked and either successfully pushed to $\mathscr{G}$ or dropped, and no termination condition has been hit, $\mathscr{G}$ becomes the new base frame and the successor frame is emptied. If at any point $\mathscr{F} = \mathscr{G}$ then $\mathscr{F}$ is a k-inductive strengthening of $\neg Bad$, proving that Bad is unreachable in the system. In addition to the two frames IcE maintains a queue Q. The queue contains the elements of $\mathscr{F}$ that still need to be processed at the current level. The elements of Q are also referred to as *obligations*.

Following is a formalization of the induction engine as a set of rules that manipulate the current state of IcE. The state of IcE, or *configuration*, is the 5-tuple $\langle \mathscr{F}, \mathscr{G}, n, k, Q \rangle$ with $\mathscr{F}$ being the base frame, $\mathscr{G}$ the successor frame, n the current level, Q the queue of obligations, and k defining the depth of induction. For brevity, elements of $\mathscr{F}$ are referred to as lemmas instead of bounded invariants. The initial configuration of IcE is $\langle \{\neg Bad\}, \emptyset, 0, 1, \{\neg Bad\} \rangle$ and IcE makes progress by applying the following rules. Note that the rules **Safe** and **Unsafe** are special, *terminating* rules.

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

$$\textbf{Push-Lemma: } \frac{\langle \mathscr{F}, \mathscr{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathscr{F}, \mathscr{G} \cup \{l\}, n, k, Q \rangle} \quad \textbf{if } \{ \ l \text{ is } \mathscr{F}^k\text{-inductive}$$

$$\textbf{Add-Lemma: } \frac{\langle \mathscr{F}, \mathscr{G}, n, k, Q \rangle}{\langle \mathscr{F} \cup \{l\}, \mathscr{G}, n, k, Q \cup \{l\} \rangle} \textbf{ if } \{ \ l \text{ is an } n\text{-invariant}$$

$$\textbf{Drop-Lemma: } \frac{\langle \mathscr{F}, \mathscr{G}, n, k, Q \cup \{l\} \rangle}{\langle \mathscr{F}, \mathscr{G}, n, k, Q \rangle} \quad \textbf{if } \{ \ l \neq P$$

Additionally, the rules **Safe** and **Unsafe** are special *terminating* rules.

$$\textbf{Safe: } \frac{\langle \mathscr{F}, \mathscr{G}, n, k, \emptyset \rangle}{SAFE} \textbf{ if } \{ \ \mathscr{F} = \mathscr{G}$$

$$\textbf{Unsafe: } \frac{\langle \mathscr{F}, \mathscr{G}, n, k, Q \rangle}{UNSAFE} \textbf{ if } \{ \ \mathit{Bad} \text{ is reachable in } [n+1, n+k] \text{ steps.}$$

The rules of IcE, namely **Add-Lemma** and **Drop-Lemma**, are abstract in the sense that it is neither specified when or how new lemmas learnt nor when they should be dropped. In sequential setting, new lemmas are typically learnt from FiRE when a counter-example to induction of some obligation is showed to be unreachable by FiRE.

The rule **Add-Lemma** is general enough to cover not only the *internal* learning, but also *external* learning. Internal learning refers to the learning of lemmas from FiRE, while external learning refers to the lemmas that 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.

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 $\mathscr{S}$ 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 $\mathscr{S}$ 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.

2.4 Smart Contracts

In ETHEREUM [Eth18a] smart contracts are conceptually similar to classes as in object-oriented programming. A contract is constructed when deployed in the blockchain, and it can be interfaced through function calls. Users and smart contracts are identified by a unique ETHEREUM *address* and interact with each other by submitting transactions to an address. The address that performs the transaction is called *sender*, while the address that receives the transaction is called *receiver*. If the receiver is a user the result of the transaction is the transfer of a specified amount of cryptocurrency. Otherwise, when the receiver is a smart contract the transaction triggers the execution of a specified contract function. During the computation, smart contracts can submit transactions to other addresses to transfer funds or recursively interact with other contracts.

The execution of the ETHEREUM platform is carried out by *miners* that mine transactions for a fee. The fee is based on the cost of the transaction as specified by the execution environment, in an abstract quantity called *gas*. The sender specifies the price for a unit of gas, and provides an amount of money for the transaction. The miner then keeps the price of the actual gas used in the transaction from the amount as a compensation for mining the transaction and returns the rest back to the sender.

Smart contracts consist of a *storage* and a set of *functions*. The storage is a persistent memory space used to store variables whose values represent the contract state. In ETHEREUM, the storage lives in the blockchain that guarantees persistency and coherence. Functions are allowed to access the storage both in read and write modes, and their behaviour is defined by the corresponding ETHEREUM *Virtual Machine* (EVM) low-level instructions, stored persistently in a separate memory residing within the blockchain. Each contract function is either *external* or *internal*. Transactions to a contract can be performed by calling only external functions. Internal functions can only be called from inside the contract during an execution.

Smart contracts can be written in several languages that compile to EVM. Solidity and Vyper are the most popular languages. Solidity is a Turing-Complete language specifically designed for smart contracts targeting EVM. The syntax is similar to C++. Solidity defines contracts as structures similar to classes in object-oriented programming languages, and natively supports several data types

and structures, e.g. integer, Boolean, arrays, maps, strings and structs.

Each external function execution, called from either the same or another contract, is identified by a separate transaction. During the execution, the entire system can *revert* to the state prior to the beginning of the transaction, that is, as if the call had not happened, before returning to the caller. Reverting is performed as follows: Given a call to a function f at program counter value c, the current state of all the variables in the scope of f, i.e. all the variables f is allowed to access, is saved as σ_f^c. In case of invoking revert, the system recursively traverses the executed call tree of f, i.e., all the functions called since the beginning of the execution of f. The visit of the call tree of f is done anti-chronologically in a depth first manner, and for each node visited representing the call to some $\hat{f}$ with program counter value $\hat{c}$, the associated state $\sigma_{\hat{f}}^{\hat{c}}$ is restored. Finally, the state σ_f^c is restored, and the execution resumes in the caller that is notified of the occurrence of the reversion. If the call is initiated by a user, then f is at the root of the executed call tree, the entire transaction is reverted, and the notification is delivered to the user. Each individual external function call is therefore an *atomic* transaction, i.e., it either executes without exceptions committing the changes, or rolls back completely if an exception occurs, leaving the state unchanged. Contrarily, in standard programming languages all the changes made in the heap by a function prior to throwing an exception are preserved.

A *control flow graph* (CFG) is a graph representation of the execution paths of a program, and it is commonly used for static analysis. A graph node represents a *basic block*, that is, a sequence of program statements that do not change the control flow of the program. Common programming language constructs that modify the control flow are branching, loops, and function calls. Moving from one block to another is a *jump*. Often jumps are conditional, i.e. they are labeled with a Boolean expression that must be true for the jump to occur.

Chapter 3

Multi-agent SMT Solving

Determining the satisfiability of a first-order formula is in general undecidable. However, the SMT approach to first-order formula solving aims at being very practical by focusing on the needs of specific user communities. This focus on user communities allows the development of algorithms that are often highly efficient in the decidable fragments of first-order theories required by users' applications. Nevertheless, state-of-the-art SMT solvers still face strong performance issues on real-world problems.

T-DPLL [Tin02, NOT06] is the standard algorithm for SMT solvers, and its execution can be described as a sequence of operations that iteratively builds a solution. Heuristics are in charge of choosing the next operation to perform during the execution, therefore their ability to *take the right choice* is of utmost importance. Small changes in heuristics reasoning induce a chain reaction that can produce substantial differences with respect to the performance.

This chapter studies how parallelism, and in particular distributed computing with cooperating multi-agents, can help T-DPLL solvers to scale to increasingly hard problems. There is a rich variety of first-order theories supported by the SMT solving community. However, the T-DPLL solvers have only three central algorithms that are employed in solving most theories: the *conflict-driven clause-learning algorithm* (CDCL) [MSS99] for propositional satisfiability used in the core SAT solver, the *congruence closure algorithm* for ground first-order logic [DNS05], and *a backtrackable Simplex algorithm* for linear real arithmetic [DdM06]. In particular this chapter focuses on instances from the quantifier-free fragments of *equality logic with uninterpreted functions* (EUF), and *linear real arithmetic* (LRA). These theories are enough to produce empirical evidence regarding the performance of all the three central algorithms while executed in a parallel cooperating settings.

3.1 Background

3.1.1 T-DPLL Algorithm

A T-DPLL [Tin02, NOT06] solver consists of a SAT solver determining the satisfiability of a CNF formula F, and a theory solver that is capable of reasoning on a conjunction of predicates over a theory T. In the pre-processing phase the input SMT formula is converted into an equisatisfiable propositional formula F in CNF while preserving the special T-interpretations of the atoms. The solving process is driven by a SAT solver maintaining a set of clauses which initially consists of the formula F. During the search the SAT solver maintains a state $\sigma \parallel F$, where σ is an initially empty ordered assignment. The state is modified according to the following rules, adapted from [BSST09]. In the following, the notation c^L and c^E respectively refer to *learned* and *explanation* clauses. Rules not having explicit labels match any clause.

$$\textbf{Prop:}\ \frac{\sigma \parallel F \wedge (c \vee l)}{\sigma l \parallel F \wedge (c \vee l)}\quad \textbf{if}\ \begin{cases} c \text{ is a clause} \\ \neg c \subseteq \sigma \\ l \notin \sigma \text{ and } \neg l \notin \sigma \end{cases}$$

$$\textbf{Decide:}\ \frac{\sigma \parallel F}{\sigma l^d \parallel F}\quad \textbf{if}\ \begin{cases} l \text{ or } \neg l \text{ appears in } F \\ l \notin \sigma \text{ and } \neg l \notin \sigma \end{cases}$$

$$\textbf{Fail:}\ \frac{\sigma \parallel F \wedge c}{\mathit{Fail}}\quad \textbf{if}\ \begin{cases} \neg c \subseteq \sigma \\ \sigma \text{ contains no decision literals} \end{cases}$$

$$\textbf{Restart:}\ \frac{\sigma \parallel F}{\emptyset \parallel F}$$

$$\textbf{T-Prop:}\ \frac{\sigma \parallel F}{\sigma l \parallel F \wedge (c \vee l)^L}\quad \textbf{if}\ \begin{cases} \sigma \models_T l \\ l \in \mathrm{At}(F) \text{ or } \neg l \in \mathrm{At}(F) \\ l \notin \sigma \text{ and } \neg l \notin \sigma \\ c \text{ is a clause s.t. } \sigma \models_T \neg c \end{cases}$$

$$\textbf{T-Explain:}\ \frac{\sigma \parallel F}{\sigma \parallel F \wedge c^E}\quad \textbf{if}\ \begin{cases} \text{each atom of } c \text{ appears in } \sigma \parallel F \\ \sigma \models_T \neg c \end{cases}$$

$$\textbf{P-Explain:}\ \frac{\sigma \parallel F}{\sigma \parallel F \wedge (c_1 \vee c_2)^E}\ \textbf{if}\ \begin{cases} c_1 \vee x \in F \text{ and } c_2 \vee \neg x \in F \\ \neg c_1 \subseteq \sigma \\ \neg c_2 \subseteq \sigma \end{cases}$$

$$\textbf{Forget:}\ \frac{\sigma \parallel F \wedge c^L}{\sigma \parallel F}$$

$$\textbf{Backjump:}\ \frac{\sigma l^d \sigma' \parallel F \wedge c^E}{\sigma l' \parallel F \wedge (c' \vee l')^L}\ \textbf{if}\ \begin{cases} \neg c \subseteq \sigma l^d \sigma' \\ \text{There is a clause } c' \vee l' \text{ s.t.} \\ (1)\ F, c \models_p c' \vee l' \text{ and } \neg c' \subseteq \sigma; \\ (2)\ l' \notin \text{At}(\sigma) \text{ and } \neg l' \notin \text{At}(\sigma); \text{ and} \\ (3)\ l' \text{ or } \neg l' \text{ occurs in } \sigma l^d \sigma' \parallel F \wedge c \end{cases}$$

A T-DPLL-based SMT solver works by applying the above rules with certain restrictions: the rule **Decide** is never applied if the rule **Prop** is applicable, the rule **Backjump** is always applied after either **T-Explain** or **P-Explain** is applied, and if the rule **Decide** cannot be applied (i.e., all atoms are assigned) the solver applies the rule **T-Prop**. The rule **Decide** adds to σ a literal l^d, that is, l labeled as decision literal.

The solver terminates by reaching a state $\sigma \parallel F$ where either σ is a total assignment proving F satisfiable, or the rule **Fail** is applicable proving F unsatisfiable. The solver always terminates if the rule **Forget** is applied with an increasing interval [BSST09].

The clauses $(c' \vee l')^L$ resulting from the application of the rule **Backjump** are *learned clauses*. The *unit propagation closure* $UP(F, \sigma)$ is defined as the set σ', where given a solver state $\sigma \parallel F$, the rules **Prop** and **T-Prop** are applied until neither one applies, resulting in $\sigma' \parallel F$. Note that $UP(F, \sigma)$ is unique, since truth assignments are sets and therefore unordered.

3.1.2 Parallelization Approaches for SMT

Heuristics for guiding the search on a Boolean structure, and in particular the choice of the literal l^d in application of the rule **Decide**, play an important role in T-DPLL solvers. During the search solvers typically maintain heuristic values for the free atoms. As a natural consequence of the complexity of the SMT problem, these values are inaccurate and small changes in them can result in significant differences in run times. To discuss the run time of a solver S on an instance F this chapter introduces the *cumulative run time distribution* $q_F^S(t) : \mathbb{R} \to [0, 1]$ that gives the probability that the solver S solves the instance F in time less

than or equal to t. We visualize such distributions in section 3.3. In some cases the distributions are *heavy-tailed* [GSCK00], having a significant probability of producing outliers, measurements that are far from the median. When such a distribution is visualized in the xy-plane, it has long, nearly horizontal segments. The randomness can be used to obtain speed-up in a parallel setting using a *portfolio* of algorithms. The main challenge in parallelizing SMT solvers this way is that portfolio-style solving seems to hit a scalability limit where adding more CPUs does not provide more speed-up [HJN11, HM12, KSSS13]. The scalability problem of the portfolio-style solving can be addressed by allowing the search processes to share information such as the learned clauses obtained with the rule **Backjump**. This chapter uses the simple portfolio obtained by forcing the rule **Decide** to pick a random literal l, potentially against the heuristic value. This approach was found to be efficient in SAT solving [HJN09] and was identified to be the best-performing strategy for SMT solvers in [WHdM09].

The scalability limit is addressed in an orthogonal way by using a divide-and-conquer approach where several solvers work in parallel on problem instances that are constructed by partitioning the search-space of the original instance and hence are different from each other. The solution to the original problem instance can be obtained by combining the results from the partitioned instances. This approach has an inherent problem that needs to be addressed to obtain good results: If the original problem instance is unsatisfiable, the variance in run time results in decreased performance. Instead of having to solve a single instance the solver needs to solve several instances that, despite being usually easier than the original, might still be challenging. While the variance in solver run time makes the portfolio approach efficient, it degrades the performance of the divide-and-conquer approach, effectively resulting in the solver having to wait for the "unluckiest" instance to be solved. Under certain assumptions it can be shown that for implementations based on pure divide-and-conquer it is possible to come up with a run-time behaviour that results in increased run time when parallelized this way, and that a different organization of the search, based on the *parallelization tree* abstract algorithmic framework discussed below, can help to avoid this problem [HM12].

To present the approach, the idea of combining divide-and-conquer with portfolio is formalized using the abstract parallelization algorithmic framework called *parallelization tree* and give five concrete instantiations of the framework. In addition to providing us with a convenient tool for discussing different parallelization algorithms, the framework is also used as a tool for explaining the performance results presented in section 3.5 .

In the following this section first discusses certain approaches for partitioning

the search-space in SMT and then describe the parallelization tree framework. Then the section concludes with concrete examples of the framework instantiations.

3.1.3 Search-Space Partitioning

The basic approach for constructing partitions in SMT uses a *partitioning function*, denoted by $partf_n : F \mapsto F_1, \ldots, F_n$, to divide an SMT instance F into n partitions $F_1, \ldots, F_n$. The function satisfies the conditions that F is satisfiable if and only if $F_1 \vee \ldots \vee F_n$ is satisfiable and no two partitions F_i, F_j, $i \neq j$, share a total satisfying truth assignment. Partitions are constructed by conjoining to F *partitioning constraints* $P_1, \ldots, P_n$, that are in general sets of clauses. This chapter uses two types of partitioning constraints: the ones obtained with the *scattering approach* [HJN11] and the ones obtained with *guiding paths* [BS96, ZBH96].

The scattering approach. The scattering approach is a technique for dividing an instance into arbitrary number of partitions. Each partitioning constraint P_i is obtained by heuristically selecting a set of *scattering literals* $l_1^i, \ldots, l_{k_i}^i$ and conjoining these literals with the clauses obtained by negations of the previous scattering literals. More formally this can expressed as $P_i := l_1^i \wedge \ldots \wedge l_{k_i}^i \wedge (\neg l_1^{i-1} \vee \ldots \vee \neg l_{k_{i-1}}^{i-1}) \wedge \ldots \wedge (\neg l_1^1 \vee \ldots \vee \neg l_{k_1}^1)$. The number of scattering literals k_i is selected so that the constructed partitions have approximately equally sized search space, under the assumption that fixing a literal will reduce the search space by a constant factor. If n is the number of partitions to be generated, it can be shown that the fraction obtained by fixing the scattering literals $l_1^i, \ldots, l_{k_i}^i$ should be $r_i = \frac{1}{n-i+1}$ of the previous instance F_{i-1} [Hyv11]. It is assumed that fixing a literal will half the search space, and this results in us choosing the number of scattering literals k_i that minimizes the difference $|r_i - 2^{-k_i}|$.

The guiding paths. As an alternative to the scattering-based method for constructing the partitions, this paragraph presents a simple variant of the *guiding path* approach [ZBH96] where a binary tree with literals as nodes represents the $n = 2^k, k \geq 1$ partitions. The root of the tree consists of the *true* literal, and the rest of the nodes are either leaves with no children or have exactly two children labeled with v and $\neg v$, respectively, where $v \in B$. Each path of literals $true, l_1, \ldots, l_k$ from the root to a leaf l_k corresponds to a partitioning constraint $l_1 \wedge \ldots \wedge l_k$.

Both the scattering and the guiding path approach rely on a heuristic in order to select the literals used for the partitioning. Two different heuristics have been implemented: the VSIDS-based heuristic [MMZ+01] which scores higher the atoms that are often involved in applying the rule **Backjump**, and the *lookahead heuristic* [ZM88, Sim00, HvM09] which scores atoms according to the number of literals propagated with rules **T-Prop** and **Prop**. The VSIDS heuristic is used together with the scattering approach for constructing partitions, while the lookahead heuristic is used with the simplified guiding path approach. When using the VSIDS heuristic we dedicate a short amount of time to perform a search on the instance so that the VSIDS heuristic gets reasonable scores for the atoms.

The lookahead heuristic starts with an assignment σ and computes for each atom satisfying the condition of the rule **Decide** the sizes of the sets $UP(F, \sigma x) \setminus \sigma$ and $UP(F, \sigma \neg x) \setminus \sigma$. The highest score is assigned to the atom that maximizes the minimum of the sizes of these two sets. As a result, the heuristic favors atoms that lead to similar sized partitions having few free atoms.

3.1.4 Clause sharing

In clause sharing the clauses learned by an SMT solver while solving a formula F are distributed among the solvers in the parallel portfolio. Since clause learning plays an important part of the T-DPLL solving process, sharing them can significantly speed up the parallel solving. Intuitively the shared clauses make it easier to produce the required clauses in case of unsatisfiability, and reduce the number of assignments the solver covers before finding a satisfying assignment.

3.2 The Parallelization Tree Framework

The key idea in obtaining well-performing parallel solvers where search-space partitioning plays a role is to combine elements from both the search-space partitioning and the algorithm portfolio.

The *parallelization tree* abstract algorithmic framework provides a unified way of presenting and comparing different parallelization algorithms. The parallelization tree consists of two types of nodes: *p-nodes* and *r-nodes*, where *p* stands for *partition* and *r* for *repeat*. The root and the leaves of the parallelization tree are *p*-nodes. Each *p*-node is associated with an SMT instance and, with the possible exception of the root of the parallelization tree, with one or more SMT solvers. The instance at the root of the parallelization tree is satisfiable if any instance in the *p*-nodes is shown satisfiable. A subtree rooted at a *p*-node

is unsatisfiable if one of its children is unsatisfiable or at least one of the solvers associated with the p-node has shown the instance unsatisfiable. A tree rooted at an r-node is unsatisfiable if every tree rooted at its children is unsatisfiable.

The *partitioning operator* $\text{split}^k(n_1, \ldots, n_k, F)$ is used to construct the parallelization tree. The result of applying the operator split^k on a p-node F is a tree rooted at the p-node F having k children $o_1, \ldots, o_k$. Each child node o_i is an r-node and has as children the p-nodes $a_1^i, \ldots, a_{n_i}^i$. Finally, each p-node a_j^i is associated with the partition obtained by applying the (randomized) partitioning function $partf_{n_i}$ on the formula F.

As instances of the parallelization tree, the following list identifies five particularly interesting parallelization algorithms.

- The *plain* partitioning approach $plain(n, F)$ corresponds to the parallelization tree $\text{split}^1(n, F)$ where each of the instances associated with the nodes $a_1^1, \ldots, a_n^1$ is solved with a single SMT solver.

- The *portfolio* approach $portf(k, F)$ corresponds to the parallelization tree consisting of the root associated with the instance F and using k SMT solvers to solve the instance.

- The *safe* partitioning approach $safe(n, s, F)$ corresponds to the parallelization tree $\text{split}^1(n, F)$ and solving each of the instances $a_1^1, \ldots, a_n^1$ with s SMT solvers.

- The *repeated* partitioning approach $rep(n, k, F)$ corresponds to the parallelization tree $\text{split}^k(n, \ldots, n, F)$ where each instance associated with the nodes $a_1^1, \ldots, a_n^1, \ldots, a_1^k, \ldots, a_n^k$ is solved with one SMT solver.

- The *iterative* partitioning approach $iter(k, F)$ corresponds to the infinite parallelization tree where every instance associated with a p-node is being solved with a single SMT solver and every p-node associated with an instance F_a has the single r-child and p-grandchildren constructed by applying the operator $\text{split}^1(n, F_a)$.

Figure 3.1 illustrates the corresponding parallelization trees and the solver assignments. When clear from the context, the formula F is omitted as well as the other parameters from the partitioning approach.

Concrete SMT instantiations of the parallelization tree include the CVC4 [BCD+11] and Z3 [dMB08] SMT solvers which implement a portfolio, and PBOOLECTOR [Rei14] which implements an iterative partitioning approach.

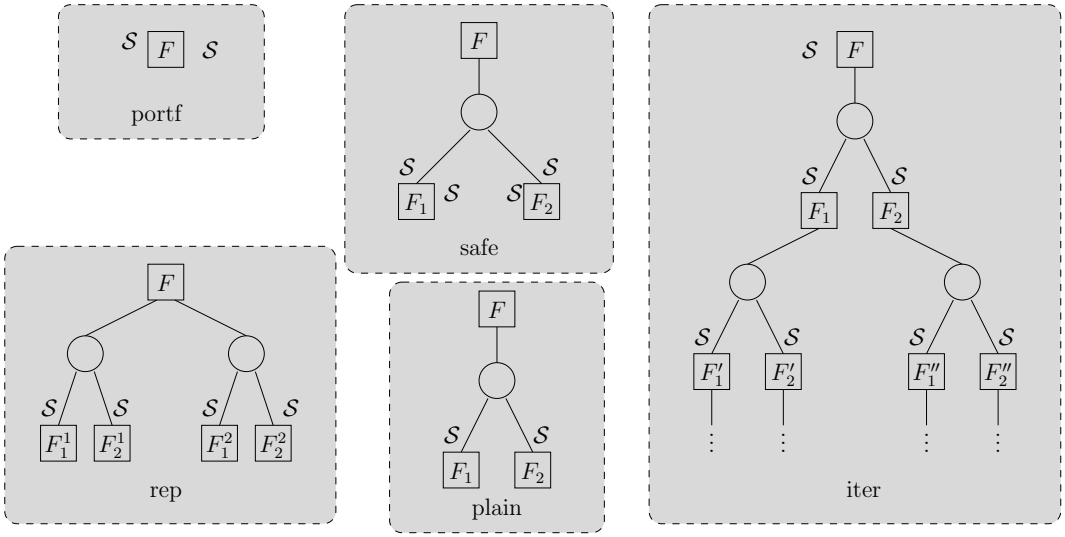


Figure 3.1. Example parallelization trees (clockwise from the top left): *portf*$(2, F)$, *safe*$(2, 2, F)$, *iter*$(2, F)$, *plain*$(2, F)$, and *rep*$(2, 2, F)$. The p-nodes are drawn with boxes, and the r-nodes with circles. The SMT solvers are indicated with the symbol $\mathscr{S}$.

3.3 Partition Distributions

This section analyses experimentally how partitioning affects the solving of SMT instances over the theories of linear real arithmetic and uninterpreted functions. The goal is to provide an insight in particular to the unexpected slow-downs and super-linear speedups by measuring the run time distributions of the original instance and the constructed partitions.

3.3.1 Linear Real Arithmetic

The benchmarks set consists of all instances from QF_LRA that are unsatisfiable and solved between 100 and 1200 seconds by OpenSMT2 [HMAS16] on a single, arbitrary run. This set results of 103 instances in the experiments. These instances were partitioned into two parts with the approach *plain* using guiding paths with the lookahead heuristic. The heuristic itself solved 15 of these instances, which left us 88 unsolved instances. The resulting two partitions were solved, and reported here is the higher of these two run times as the wall-clock run time. The experiment corresponds to a simulation where partitioning is assumed to take no time and there are no communication delays.

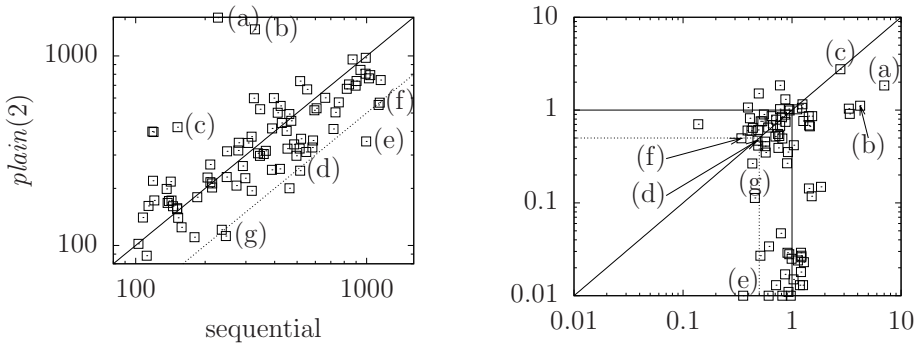


Figure 3.2. Comparison of OpenSMT2 and *plain* for QF_LRA benchmarks (left), and relative run times of the partitions with respect to the original instance (right). The dotted line represents linear two-fold speed-up, and the labels (a) – (g) refer to instances that are analyzed in fig. 3.3 more in detail.

Figure 3.2 (*left*) reports the results. The majority of the problems, 59%, are solved faster in this experiment with parallelization, and approximately 7% of the instances show super-linear speed-up. However, there are at least five instances which seem to show a significant slowdown with this approach.

Figure 3.2 (*right*) shows the run times of each of the two partitions relative to the original instance solved sequentially. Each point corresponds to an instance and has as coordinates the ratios between the solving time of the partition and the original instance. Points having both coordinates less than one and less than half (inside the solid and dashed squares, respectively) represent, respectively, instances that are faster and superlinearly faster to solve with *plain*. In the case of the serious slowdowns, both partitions were solved slower than the original instance. The super-linear speedup of instance (e) shows a significant imbalance between the partition hardness. In general the partitions seem to be relatively balanced in difficulty: for example the cases where one partition is ten times faster to solve than the original instance constitute roughly one third of the cases.

Distribution analysis. First, it is analysed the cases where *plain* takes more time than the sequential solving, using as case studies three outliers in fig. 3.2: instance (a), where the run time was almost 1600 seconds for *plain* but only slightly more than 200 seconds for OpenSMT2; instance (b) where the run time was slightly less than 1400 seconds for simulated partitioning and less than 400 seconds for simulated partitioning; and (c) where both partitions where solved in more than twice the time needed by OpenSMT2. The experimental run time

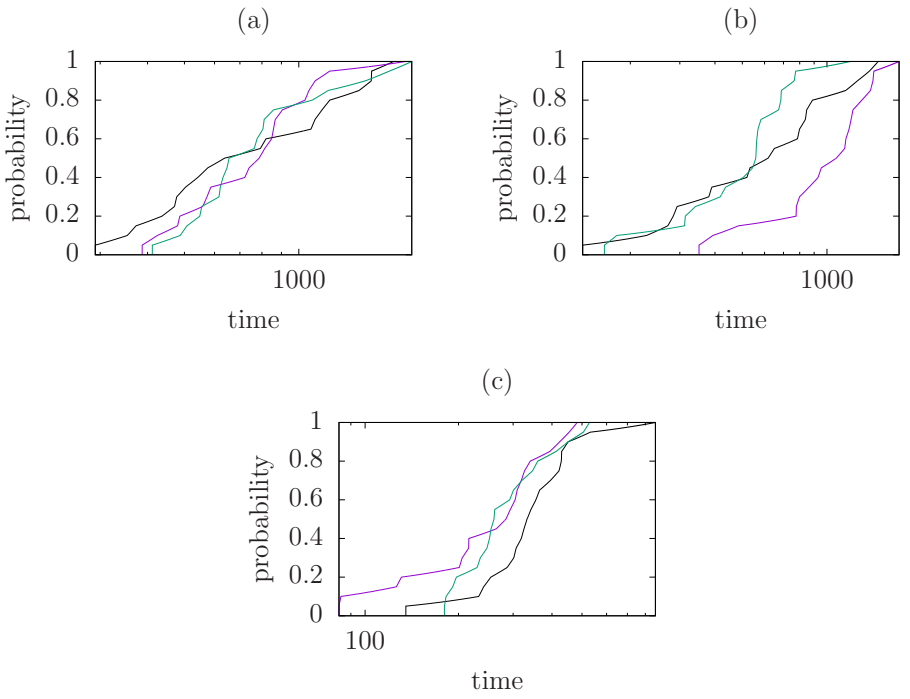


Figure 3.3. Run time distributions of three instances (a) – (c) from fig. 3.2 that perform unexpectedly bad in partitioning.

distributions reported in fig. 3.3 were obtained by running the original instances (a) – (c) and their respective two partitions 20 times seeding differently the randomness of the solver. The figures show the probability $q(t)$, on the vertical axis, that an instance is solved within time t, running on the horizontal axis. Three kinds of behavior can be observed.

- Figure 3.3 (a): Both partitions are roughly equal to the original instance. The high run time is due to having to wait for the longest-running instance.
- Figure 3.3 (b): one of the partitions has a consistently lower probability of being solved within time t compared to the original instance. The high run time in fig. 3.2 can be explained by this harder partition.
- Figure 3.3 (c): the resulting distributions are consistently higher for the two partitions than for the original instance, and the high run time in fig. 3.2 is a result of "bad luck".

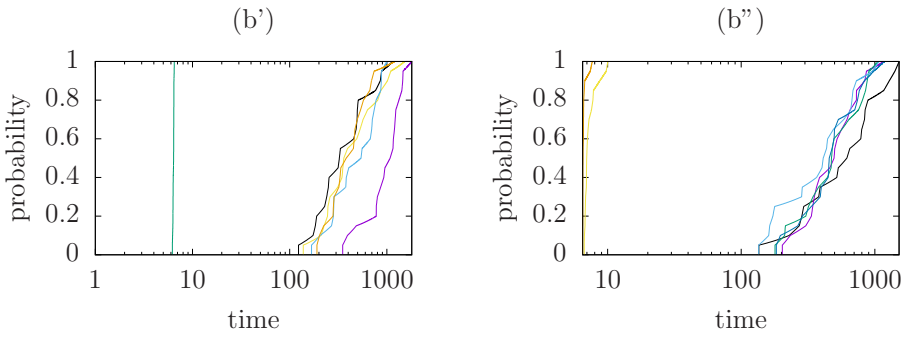


Figure 3.4. Run time distributions of the instance (b) from fig. 3.2 when partitioned into four (left) and eight (right).

Figure 3.4 (b') and (b") further shows run time distributions for the outlier (b) when partitioned into four and eight parts. Figure 3.4 (b') gives as reference the hard distribution from (b) in purple. The partitioning resulted in three distributions that, while clearly closer to the original instance, are still somewhat harder, in addition to an easy partition solvable in six seconds. Partitions seem to be inherently harder than the original in this instance. When partitioning to eight parts in fig. 3.4 (b") all the partitions seem to already be easier than the original instance in average.

Similar to the unexpected slow-downs, a super-linear speedups is observable in fig. 3.2. In the experiments a total of six instances present super-linear speedup where both partitions were solved in less than half the time required to solve the original instance. Figure 3.5 analyses closer four representative examples, marked in fig. 3.2 by labels (d) – (g).

- Figure 3.5 (d): One partition is clearly easier than the original instance. The original instance has a heavy tail, which is made lighter in the other partition. The speed-up could be said to be because of the lighter tail of the other partition.

- Figure 3.5 (e): One partition is very easy and the other partition is essentially the same as the original. The speed-up could be said to result from a purely random behavior.

- Figure 3.5 (f): Both partitions are harder than the original, but the partitions have heavy tails. Similar to (e), the speed-up could be said to result from a purely random behavior.

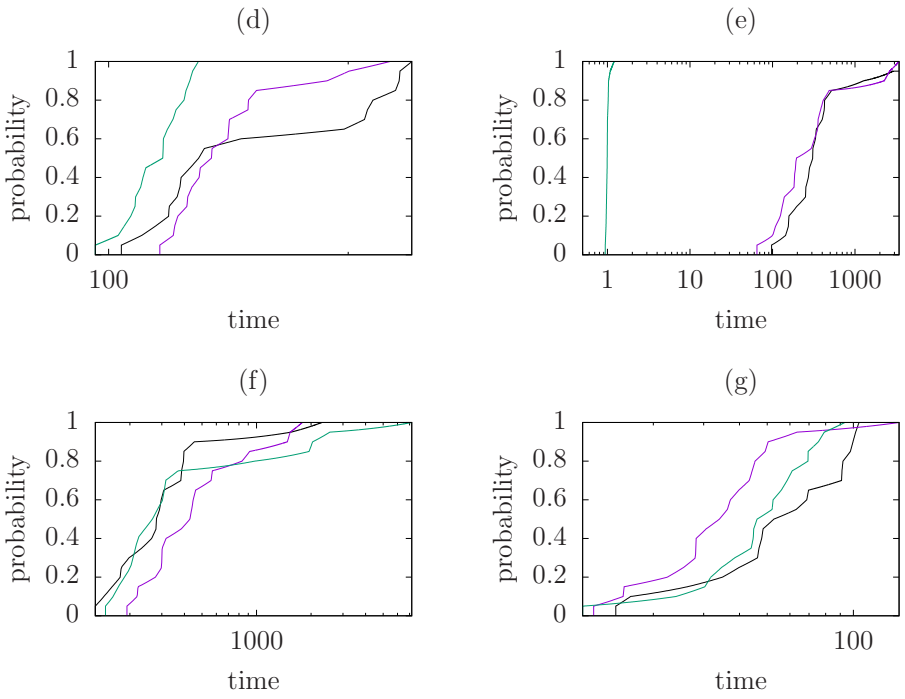


Figure 3.5. Run time distributions of four positive outliers from fig. 3.2.

- Figure 3.5 (g): Both partitions are easier than the original instance. Obtaining speed-up is therefore likely, but the super-linear behavior is random behavior.

The conclusion is that both the super-linear speed-up and slow-down effects measured in partitioning-based parallel approaches result from rich interaction between partitioning and inherent randomness in solving instances from QF_LRA. Luck often plays a major role in both slower-than-expected and faster-than-expected instances. Perhaps the most surprising effect is the consistent slow-down observed in partitions for instances (f) and (b), where the assumed literal in some runs seems to distract the solver from finding proofs quickly. We note the absence of cases where both partitions would be considerably harder than the original instance. In all six cases of super-linear speed-ups and some of the cases where slow-down is observed, the distributions exhibit long almost horizontal phases indicative of heavy-tailed behavior. This common phenomenon in constraint solving seems to play a major role in the unpredictability of the efficiency of parallel solving based on partitioning.

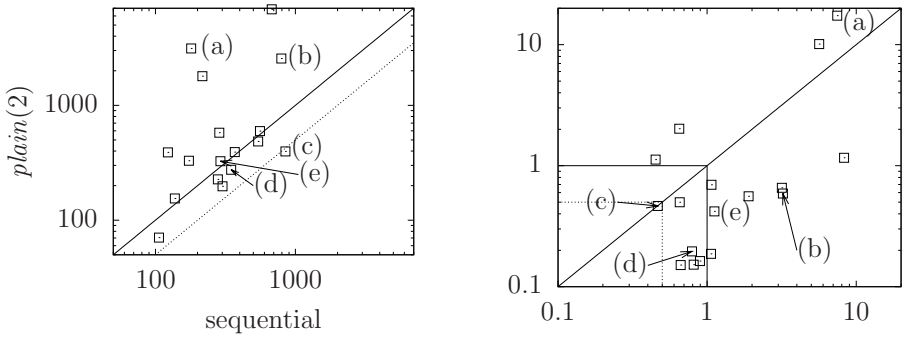


Figure 3.6. Comparison of OpenSMT2 and *plain* for QF_UF benchmarks (left), and relative run times of the partitions with respect to the original instance (right). The dotted line represents linear two-fold speed-up, and the labels (a) – (d) refer to instances that are analyzed in fig. 3.7 more in detail. The instance (e) is analyzed separately in fig. 3.8 (right).

3.3.2 Equivalence Logic with Uninterpreted Functions

Further, this thesis studies the run time distributions for instances from the category QF_UF of SMT-LIB. The instances where OpenSMT2 was able to find a solution between 100 and 1000 seconds are partitioned into two. The benchmark set obtained this way consists of 16 instances, which is too few to draw conclusive results, but nevertheless allows us to study similar interesting behavior observed for QF_LRA.

The comparison between the sequential OpenSMT2 and *plain* run times is shown in fig. 3.6 (*left*), and fig. 3.6 (*right*) shows the relative hardness of the partitions compared to the original instance. It is observable that in comparison to QF_LRA, the algorithm *plain* performs poorly in QF_UF: only one third of the instances shows a speedup and there are several outliers. Furhter, it is observable that there are case where one partition is much harder and the other much easier than the original instance. We note also one case of superlinear speed-up.

Distribution analysis. Considering the run-time distributions for the instances, four different classes of instances can be identified based on the distributions. Representative examples are marked with labels (a) – (d):

- Figure 3.7 (a): Both partitions are significantly more difficult than the original instance;

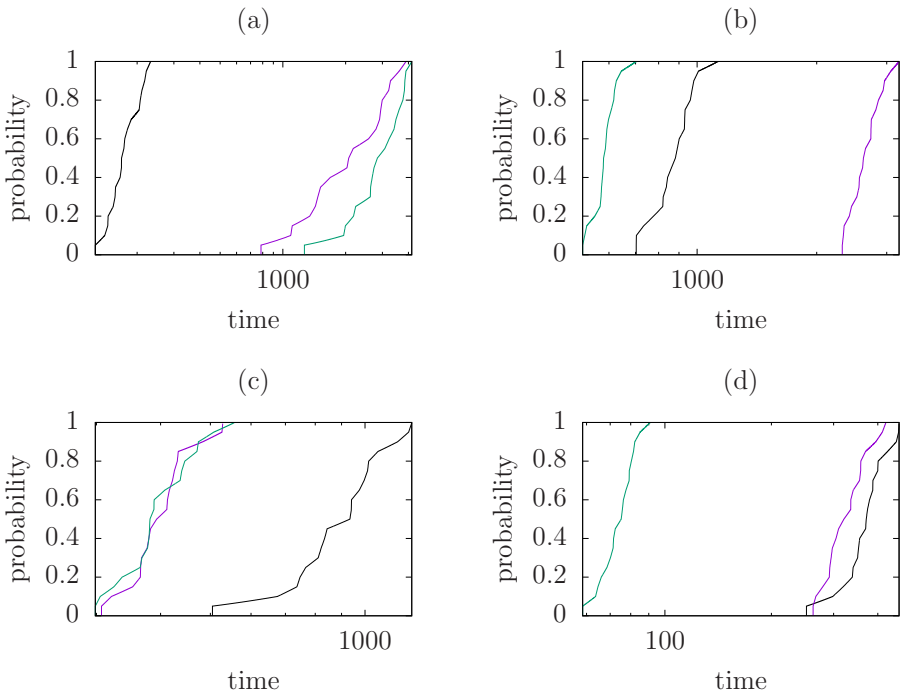


Figure 3.7. Run time distributions of four instances from fig. 3.6.

- Figure 3.7 (b): One partition is significantly more difficult than the original instance, while the other is clearly easier;
- Figure 3.7 (c): Both partitions are significantly easier than the original instance; and
- Figure 3.7 (d): One partition is significantly easier than the original instance, and the other partition is only slightly easier.

The rest of the instances fell into one of these four categories. The two new behaviors that were absent from the case for QF_LRA can be observed: a consistent increase in difficulty for both partitions in the instance (a), and a consistent super-linear speed-up for both partitions in the instance (c). For now a good explanation for these anomalies is not available. The introduced unit clauses are possibly confusing the heuristic to search for proofs in a redundant way, and since the distributions are fairly resilient to randomness in comparison to linear real arithmetic distributions, the bad behaviour is difficult to escape by increasing randomness.

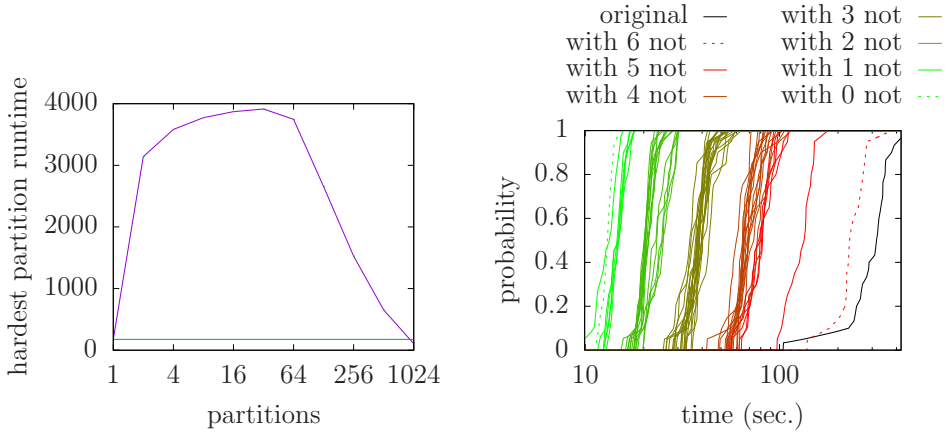


Figure 3.8. Run time with respect to number of partitions for instance (a), green line represent original instance runtime (left), and the cumulative runtime distribution of the QF_UF benchmark (h) and its 64 partitions (right).

The instance (a) is studied further by splitting to two, four, eight, 16, 32, 64, 128, 256, 512, and 1024 partitions and report the results in fig. 3.8. The increase in difficulty is consistent until it reaches a peak at 32 partitions after which the instance gets easier to solve. However, considering all splitting from 2 up to 512 partitions, the hardest partition is still harder than the original instance. Such behaviour seems to gradually recede and eventually reverse. Indeed all partitions resulting splitting in 1024 are is easier than the original instance. In the experiments, the proportion of partitions that are harder than the average run time of the original instance decreases dramatically with the increase of partitions. For example, the proportion at 32 partitions is close to 50%, for 128 it is 13%, for 256 partitions 7%, and for 512 partitions just over 1% of the partitions are harder than the average.

Finally, fig. 3.8 (*right*) shows the cumulative runtime distributions of the QF_UF benchmark instance (e) and its 64 partitions created using lookahead. In this case the partitions constraint the original instance by adding all the combinations of true/false assignments to 6 atoms. The figure shows that in this case there is a strong correlation between the number of false assignments in the partition constraints and the runtime. The instance seems to get substantially easier when atoms are asserted true. The speedup obtained by solving all 64 partitions of fig. 3.8 (*right*) in parallel is 21%. Specifically, the average CPU time for solving the original instance sequentially is 304 seconds, whereas the maximum average runtime for all the partitions is 241 seconds. However, solving all 64 partitions

would require on average 3034 CPU seconds, which is roughly 10 times more than the CPU time needed by the original instance to be solved sequentially.

As a conclusion, the algorithm *plain* performance depends in general on the underlying SMT theory, and the strategy is prone to catastrophic failures where partition gets significantly harder than the original instance. The following studies different approaches for overcoming these issues.

3.3.3 Partition Heuristics

Since the ability of the partitioning to construct easy instances plays such a critical role in the overall success of the partitioning based parallelization algorithms, it is interesting to study the effect of partitioning heuristics. Two different types of heuristics are compared, the VSIDS [MMZ+01] scattering and lookahead [ZM88] based on the guiding path implementation in [HMS+18].

The algorithm *safe*$(8, 8)$ and a superset of the previous experiments containing unsatisfiable QF_UF instances[1] are used to illustrate the difference between the two heuristics and report the results in fig. 3.9 (*top*). Excluding the time to construct the partitions, the lookahead gives a 40% reduction in the run time of the solver, the average speed-up being two. However, when the time required to construct the partitions is included, the lookahead-based heuristic looses the edge and becomes slightly worse compared to the VSIDS-based heuristic. This results mainly from the implementation of the lookahead-heuristic. The current implementation is not as optimized as the VSIDS implementation, but we believe that the heuristic can be made more efficient.

To understand the impressive efficiency of the lookahead heuristic we study closer two examples where the abstract parallelization algorithm *safe*$(2, 32)$ performs well with the lookahead heuristic and with the VSIDS heuristic. The graphs on the bottom of fig. 3.9 report the cumulative run-time distributions of the original instance and the four partitions constructed with the two heuristics. In the first example the lookahead heuristic finds a partitioning where the two partitions have a very similar run time distribution, whereas the VSIDS heuristic results in a very uneven distribution where one partition is significantly easier to solve than the other. In the second case (lower right graph in fig. 3.9) the lookahead heuristic performed on a single run worse than the VSIDS heuristic. In this case both the heuristics resulted in very uneven partitions. However it would seem that the cumulative run-time distribution of the VSIDS heuristic dominates on a wide area the distribution of the lookahead-based heuristic. Interestingly there seems

[1] All satisfiable instances were easy and the results therefore not representative.

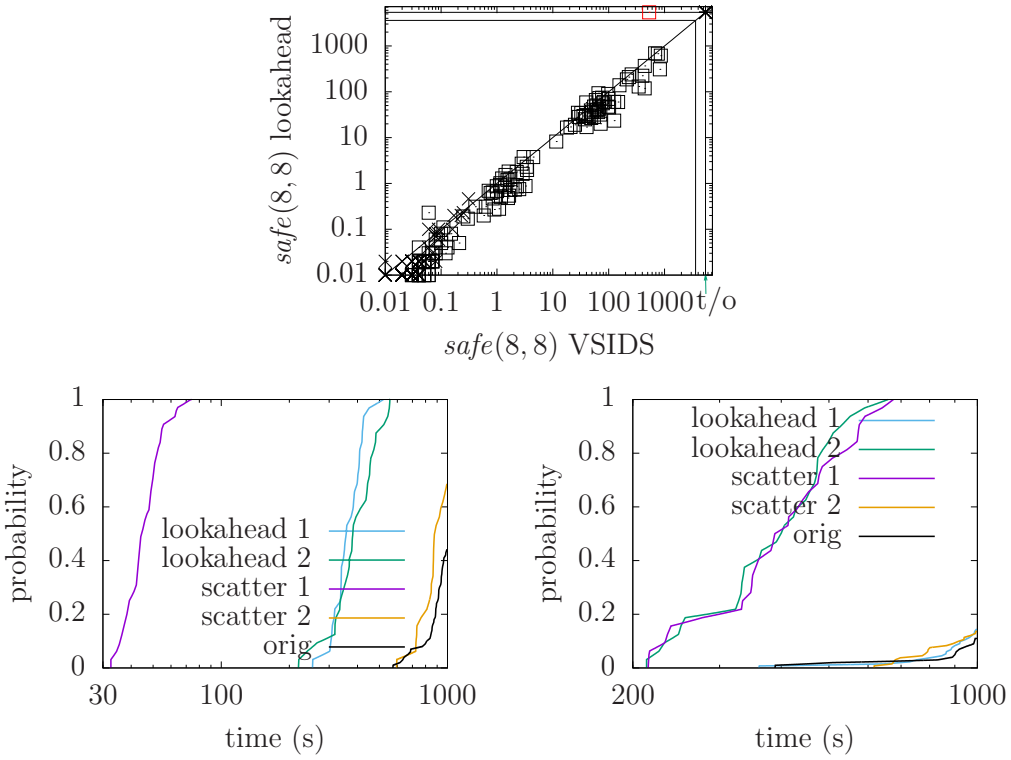


Figure 3.9. The lookahead heuristic compared to the VSIDS-based scattering heuristic

to be a small probability that the lookahead-heuristic can solve the problem somewhat faster than the original problem, suggesting that the implementation with load balancing should be capable of performing well on this instance also for the lookahead heuristic.

3.3.4 Repeated Partitioning

Due to the heuristic nature of the partitioning it is natural to ask how a portfolio of partitioning algorithms would work. Figure 3.10 provides a comparison between the algorithms *plain*(64) and *rep*(2, 32) using instances from QF_UF. The experiment set contains all instances having run time longer than one minute with the default configuration of OpenSMT2. This set consists of 54 instances, 11 of which could not be solved within the 1000 seconds timeout and 4 GB memory limit. All the instances solvable from this set turned out to be unsatisfiable, and therefore a randomly selected 100 easier satisfiable and unsatisfiable

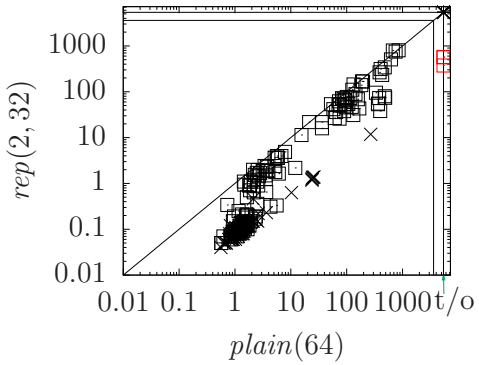


Figure 3.10. The run times for the parallelization algorithms *plain*(64) and *rep*(2, 32) on 64 cores.

instances are added, resulting in total 254 benchmark instances. The run times include the time required for the partitioning. Here, as well as in all other similar graphs, satisfiable and unsatisfiable instances are denoted respectively by × and by ⊡, and timeouts are highlighted with a red color. The algorithm *plain*(64) is almost always worse. A closer analysis reveals that when considering only the instances that both approaches could solve the algorithm *rep*(2, 32) solves the full problem set 9 times faster than the algorithm *plain*(64).

3.4 Evaluation of the Parallelization Tree Framework

This section presents the results of some of the algorithms obtained from the parallelization tree algorithmic framework of section 3.1.2 with scattering and the VSIDS heuristic. The results are presented separately for instances from the QF_UF and QF_LRA benchmarks. The reported times include also the time to run the partitioning.

3.4.1 Logic of Equality and Uninterpreted Functions

Table 3.1 shows the results for the hardest instances in the QF_UF benchmark set. The column OpenSMT2 represent the sequential run of the OpenSMT2 solver. The best run time for a given instance is shown in boldface, and dashes indicate timeouts.

While the parallelization algorithm *portf* works relatively well, it seems to lose in the hard instances when compared to the approaches that combine el-

Table 3.1. Instances solved with at least one of the approaches, but where the portfolio approach required over 100 seconds with 64 CPUs. All the instances are unsatisfiable.

Name	OpenSMT2	*portf* (64)	*rep* (2,32)	*safe* (2,32)	*plain* (64)	*rep* (8,8)	*safe* (8,8)
PEQ003_size9	437	299	336	**232**	431	286	248
PEQ004_size8	124	117	109	110	125	**108**	115
PEQ011_size8	572	302	267	**265**	388	309	280
PEQ012_size6	—	—	456	**507**	621	574	532
PEQ014_size11	737	**338**	482	564	—	—	540
PEQ016_size6	223	181	158	168	188	**158**	176
PEQ018_size7	192	144	155	**139**	182	207	218
PEQ020_size6	511	409	379	**314**	405	401	371
SEQ005_size8	174	159	144	144	132	148	**131**
SEQ010_size8	244	190	**123**	166	196	157	155
SEQ026_size7	890	708	731	794	**671**	774	725
SEQ038_size8	—	826	903	751	751	**745**	792
NEQ006_size6	—	—	—	—	—	—	—
NEQ016_size8	774	616	682	575	—	625	**419**
NEQ023_size7	—	—	—	—	—	—	—
NEQ032_size6	—	830	407	**373**	—	836	865
NEQ048_size8	476	430	421	**341**	479	349	445
NEQ048_size9	—	815	**759**	804	849	832	833
Total solved	12	15	16	16	13	15	16

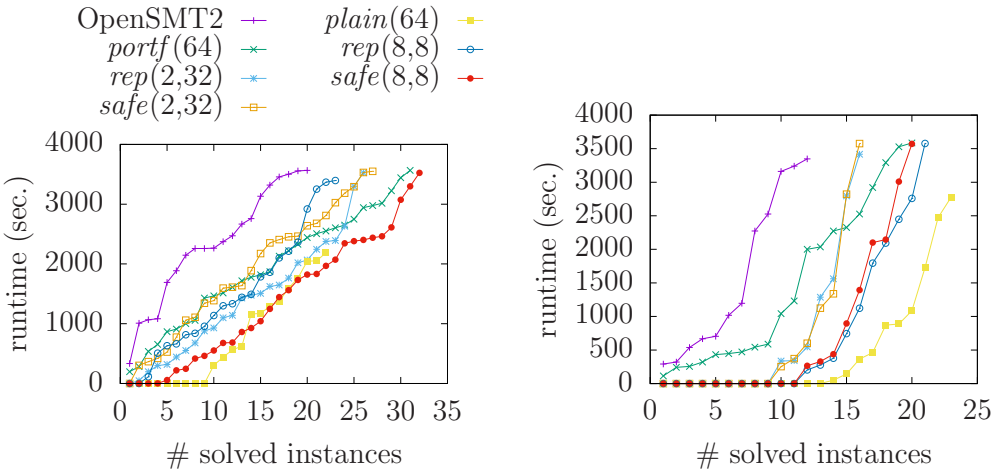


Figure 3.11. Comparison of partition tree algorithms from satisfiable (left) and unsatisfiable (right) benchmarks from QF_LRA.

ements from portfolio and search-space partitioning. In fact if the sequential execution and the algorithm *plain* are not considered, other algorithms perform better than this implementation of the portfolio. Despite the anomalies related to partitioning reported in the previous section, it seems that through the algorithms obtained from the partition tree framework it is possible to successfully incorporate partitioning to parallel SMT solving.

3.4.2 Logic of Linear Real Arithmetic

Figure 3.11 reports a similar comparison on the partition tree algorithms on instances from QF_LRA. The instances were selected so that their average run-time on OpenSMT2 was over 100 seconds. In general the chosen partition tree algorithms outperform the sequential solver, an observation that is not trivial from the initial analysis of the distributions. The results suggest that the portfolio approach performs well for both satisfiable and unsatisfiable instances, while *plain* is still slightly better on the unsatisfiable algorithms. The experimental data confirms that the approaches *rep*$(2, 32)$ and *safe*$(2, 32)$ that produce few partitions do not perform as well as the other approaches.

As a conclusion, there is a difference in the behavior of the solver with respect to parallel algorithms on the two studied logics: for QF_UF the winner on the benchmark set is *safe*$(2, 32)$, while for QF_LRA the winner is *plain*(64) for unsatisfiable, and *portf*(64) for satisfiable instances.

3.5 Clause Sharing

This section describes the experiments obtained with the parallelization tree approach using the SMTS framework [MHS18], in particular studying the effect of clause sharing. The details of the framework can be found in chapter 5. The experiments concentrate on four topics: Section 3.5.1 demonstrates how the clause sharing works on *portf* and *safe*; Section 3.5.2 reports how the filtering heuristic affects the performance of the parallel algorithms; and Section 3.5.3 compares the cloud-based implementation against a sequential version of OpenSMT2 and a widely used reference solver Z3 [dMB08].

The hardware configuration is kept the same in all experiments. The experiments were run in a cluster where with eight compute nodes for the clients and the head node for the server. Each compute node is equipped with two CPU Quad-Core AMD Opteron 2384 and 16GB of RAM. During the experiments each cluster node had eight client processes implementing the SMT solver OpenSMT2, resulting in total of 64 solvers in the entire cluster. The memory available is not explicitly limited in the solvers. The timeout is fixed everywhere to 1000 seconds. The search-space partitioning heuristic, when used, is the scattering approach [HJN06].

This section reports on a fixed benchmark set obtained from the SMT-LIB benchmarks repository [BdR$^+$10] and the QF_LRA and QF_UF logics. The set from QF_LRA was created by selecting the instances with an average sequential execution time above 100 seconds (including those in timeout) using OpenSMT2; the set consists of 106 instances in total. The benchmark set for the QF_UF is the same used in section 3.4. x The figures report algorithm names from section 3.1. The label *CS* indicates that clause sharing is used.

3.5.1 The Effect of Clause Sharing

The first experiments show how sharing clauses affects the solving time using different partitioning methods for QF_LRA (fig. 3.12) and QF_UF (fig. 3.13). For both figures the graph on the top shows that the parallelization algorithm *portf* benefits most from clause sharing: with both theories it gives a 2.05 times speedup, as well as one more QF_LRA instance and nine more QF_UF instances solved within the timeout compared to not using clause sharing.

With both theories *safe* performs worse than *portf*: the speedup due to clause sharing is 1.97 for *safe*$(2, 32)$, and speedup of 1.67 for *safe*$(8, 8)$.

To some extent these results are expected, since the number of learned clauses available inside the clause database for a single portfolio is bigger when there are

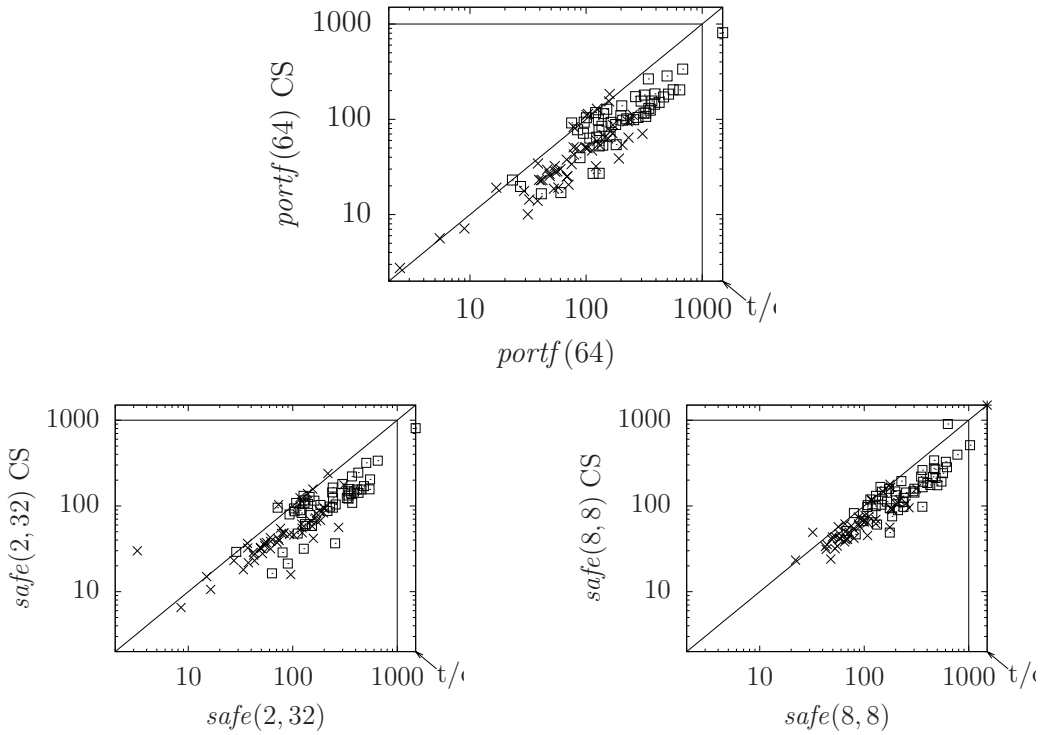


Figure 3.12. Using clause sharing against not using clause sharing with 1, 2, and 8 partitions. Framework run with 64 solvers on the QF_LRA benchmark set

more solvers running in the portfolios, and therefore also the quality of clauses that the heuristic picks is higher.

3.5.2 The Clause Sharing Heuristics

Figure 3.14 shows that clause sharing heuristics are very important: the experiment performed using a filtering heuristic that discards clauses with more that 30 literals results in clause sharing having 1.12 times greater run time compared to the run without clause sharing. Interestingly the same heuristic is working well for QF_LRA (used in fig. 3.12). To obtain good results for the benchmark instances in QF_UF the heuristic needs to be more restrictive. Reducing the threshold to 10 literals still leads to worse performance (results not shown), and discarding clauses with five or more literals gives the results on fig. 3.13.

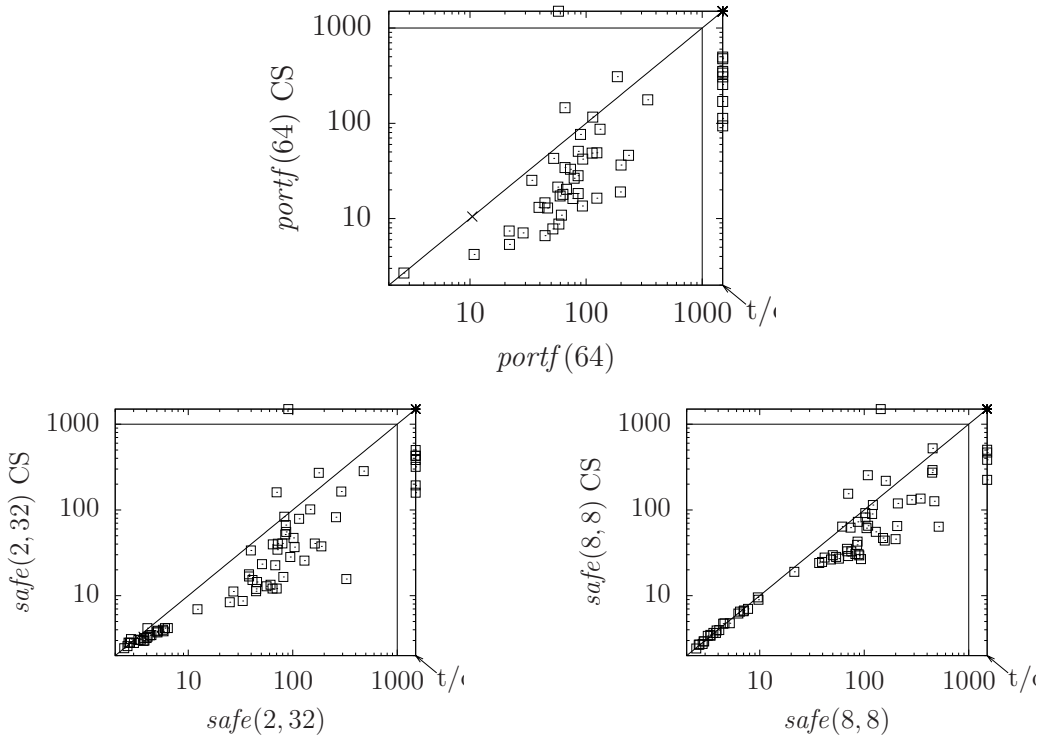


Figure 3.13. Using clause sharing against not using clause sharing with 1, 2, and 8 partitions. Framework run with 64 solver on the QF_UF benchmark set

3.5.3 Comparison to Other Solvers

Figure 3.15 compares the best known configuration of the framework against the solvers OpenSMT2 and Z3 for QF_LRA (*top*) and for QF_UF (*bottom*). The results are very promising when compared to OpenSMT2. For instances with sequential run time higher than one second and for which neither the sequential or the parallel solver timed out the average case speed-up is 4.78 for QF_LRA and 4.01 for QF_UF. The implementation is not yet competitive against Z3 in the majority of instances. This is due to the lack of optimizations in the underlying solver. Based on the experimental evidence presented in this section it seems reasonable that if either the optimizations available in Z3 were implemented in OpenSMT2 or the approach presented in this work were implemented in Z3 the results would be similarly promising in comparison to Z3.

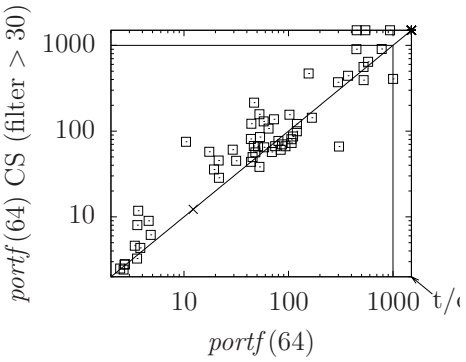


Figure 3.14. Using clause sharing with a loose filtering heuristic against not using clause sharing. Framework run with 64 solvers on the QF_UF benchmark set.

3.6 Related Work

This chapter provides an orthogonal view to the more theoretical study presented in the Handbook of Parallel Constraint Reasoning in its chapter on SMT solving [HW18] by offering a detailed experimental analysis both using controlled experiments and a computing-cluster-based implementation. The portfolio approach combined with clause sharing has been implemented using the SMT solver Z3 [WHdM09]. The implementation provides an efficient clause sharing strategy in a shared memory setup using lockless queues that hold references to the lemmas that a solver core wants to export. The experimental evaluations show that clause sharing leads to a substantial speedup on benchmarks from the QF_IDL logic. In contrast to this work, the techniques presented in this chapter support two SMT theories (QF_UF and QF_LRA), and exploit the advantages of combining portfolio with clause sharing and search-space partitioning. Moreover this implementation is designed to run in distributed computing clusters in addition to a single machine. Similarly to Z3, the SMT solver CVC4 [BCD+11] supports a portfolio-style parallel solving. Unlike this approach, the approach used in CVC4 is designed to run in a single computer and does not implement clause sharing.

A divide-and-conquer approach for the quantifier-free bit-vector logic has been implemented on top of the SMT solver Boolector [Rei14]. A portfolio parallelization approach for the logic of quantifier-free bit-vectors and bit-vector arrays is presented in [PC13]. Compared to these, this chapter differs in the supported theories and in that this work support cloud computing and are not

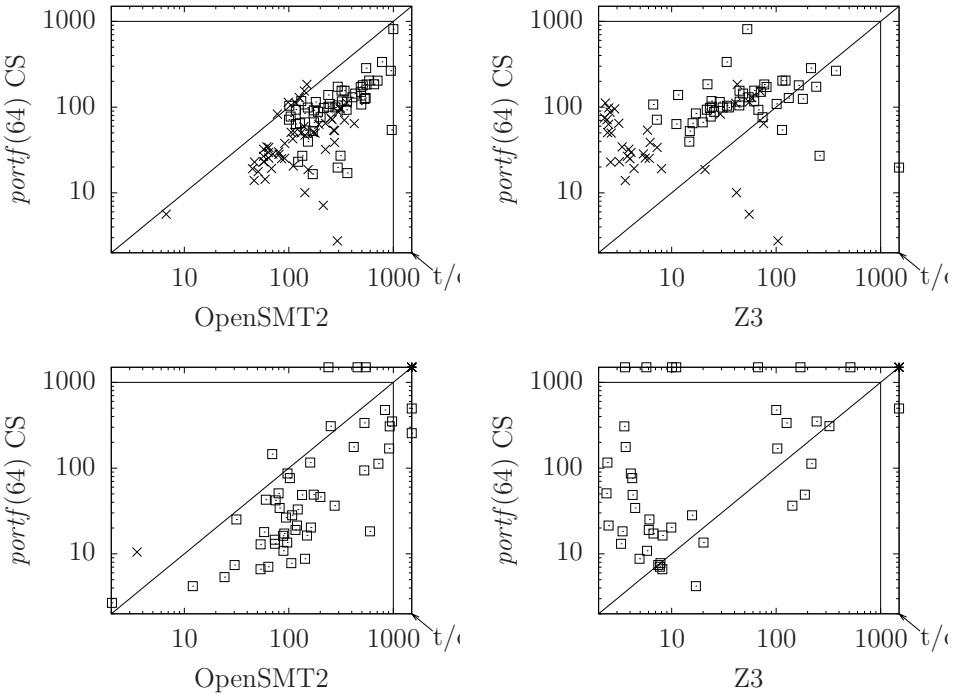


Figure 3.15. The best configuration against OpenSMT2 and Z3 for QF_LRA (top) and QF_UF(bottom)

limited to pure divide-and-conquer or portfolio approach.

In some of the experiments, this work applies a heuristic technique known as *lookahead* [ZM88] for identifying partitions that have small search space. The lookahead implementation used in this chapter is an extension of the one reported in [HMS+18]. Lookahead technique has previously been implemented for SAT in [HJN10] and in [HKWB11], and has recently been implemented in the SMT solver Z3. However, to the best of our knowledge the lookahead implementation is the first that is truly built into the T-DPLL framework and is aware of general theory propagation as well as the search space reduction resulting from learned clauses.

This work uses techniques similar to those used in parallel SAT solving. The more elaborate problem descriptions of SMT constitute a significant theoretical and engineering challenge for parallelization. In addition the use of SMT allows extending these techniques to a different domain. Given the close relation of the topics there is a significant amount of relevant research on parallel SAT solving, overviewed for instance in [MML12]. In particular, the portfolio

approach combined with clause sharing implemented in ManySAT [HJS09] and HordeSAT [BSS15]. This work extends the results of [AHJP14] in combining search-space partitioning and clause sharing in SMT.

3.7 Conclusions and Future Work

The results presented in this chapter have been published in [MHS16, HMS15]. SMT solving in distributed environments so far has received relatively little attention from the community developing and researching SMT solvers. This chapter identifies key components of parallelization approaches and provides an extensive evaluation on their effects on efficiency of multi-agent cooperative SMT solving. Specifically, the contributions reported in this chapter are: the parallelization tree algorithmic framework that allows a flexible combination of portfolio, partitioning and clause sharing; the analyses in the form of case studies of the phenomena where partitioning produces harder instances; the evaluation of the parallelization tree framework over the theories of uninterpreted functions and linear real arithmetics; the effect of clause sharing with the parallelization tree framework and different sharing schemes; the comparison with the state-of-the-art SMT solver Z3. The empirical results reported show that SMT solving can benefit significantly from parallelization, but especially QF_UF is sensitive to the heuristic used for selecting clauses to be shared.

This work opens several avenues for future directions. The following list provides three research challenges that arise directly from the results and the empirical evaluations presented in this chapter. Theory-specific aspects are a central point in all the challenges below. This underlines the particular interest of devising *theory-aware* parallel techniques to improve reasoning capabilities.

- It is little understood why fixing atoms makes the partitions sometimes harder for T-DPLL solvers, and whether and how such atoms could be identified. Possible directions to shed light on this phenomena might involve investigating the effects of partitioning with respect to the theory structure. In particular, such theory-aware partitioning can be achieved by studying how assuming a predicate changes the theory search-space, considering the domain-specific aspects of the theory. In this way, it would be possible to rank predicates based on whether the resulting partitions are balanced, or on how the complexity of the tasks reserved for the theory solver varies having the new constraint. An example of theory-aware partitioning for LRA and LIA (linear integer arithmetics) involves reasoning

on hyper-plane subspaces to reduce the dimensions of the search-space. Particular attention is needed to perform such reduction is a balanced way.

- How to leverage clause sharing with (theory-aware) partitioning. This chapter's results show that the purely portfolio based approach *portf* performs better than *safe* which applies partitioning as well, when clauses are shared. This question centres on whether it is possible to combine partitioning with clause learning effectively. Like the previous point, the theory-aware direction seems potential. In particular, different sharing strategies prioritazed by ranking learned lemmas based on whether they are *purely theory*, *mixed* or *propositional* clauses might reveal their individual effect over the current solving process in other solvers. This assumption is based from the fact that theory-valid clauses are often the result of a substantial amount of computation performed by the theory solver. Understanding how to reuse such effort effectively is of great interest

- The experiments of this chapter focus on two fundamental algorithms underlying modern T-DPLL solvers: the congruence closure algorithm for QF_UF and the Simplex algorithm for QF_LRA. Both have very different characteristics on how the parallelization should be done to obtain maximal performance. It is unclear how the approach works on the *non-convex* SMT theories such as LIA, or combinations of different SMT theories. Such hard theories would also allow the propositional-logic-based partitioning heuristics to be made more theory-aware, as there is more interaction between the SMT theory and the propositional encoding.

Chapter 4

Multi-agent Solving by Induction

Applying model checking to realistic, complex systems is highly non-trivial in part due to the computational difficulty imposed by the underlying decision problems. In particular, unbounded model checking algorithms search for violating executions within increasing finite bounds, while trying to generalize bounded safety using induction. If the generalization succeeds then a bounded proof is guaranteed to prove unbounded safety. The solving task involves several operation entrusted to different heuristics, each contributing to the overall performance. As a result, algorithms are very sensible to small tweaks in the heuristics and their performance is unpredictable.

This chapter concentrates on multi-agent cooperating techniques for the distributed execution of unbounded model checking algorithms inspired by IC3 [Bra11]. The IC3 algorithm is a relatively recent procedure that, given a transition system and a safety property, computes a safe inductive invariant or finds a counterexample for the safety of the system. The safe inductive invariant is gradually devised from the reachability lemmas learned during the search for bounded property violations.

The P3 (Parallel IC3) algorithm introduced in this chapter allows the instantiation of different parallel configurations to enhance diversification and cooperation among agents. Each configuration is evaluated thoroughly to discuss positive and negative aspects using the massive amount of 7.6 CPU years of computational power from a distributed computing environment. The P3 algorithm implements diversification among agents in a portfolio of sequential IC3 implementations by different search strategies and by randomizing the search heuristic of the underlying SMT solver. In this way, agents compute different reachability frames by learning different reachability lemmas. Cooperation is achieved by exchanging such reachability lemmas among the solvers. Finally, P3 implements

the novel partitioning approach based on the transition function of the input system and its pre-images, which are distributed among the agents as new safety properties representing a sub-problem of the original instance. As a result, each agent is focused on solving a particular sub-problem, and cooperation among all the agents is allowed without any restrictions on the individual solving task.

4.1 Background

This section gives a high-level overview of IC3/PDR algorithm. Further details of the original algorithm and its extensions to SMT can be found in [Bra11, EMB11, HB12, GI15, KGC16].

Definition 5 (IC3 Trace) *Given an instance $\mathscr{S}$ of the safety problem, an* IC3 trace *for $\mathscr{S}$ is a sequence of frames $\mathscr{F} = \langle F_0, F_1, \ldots, F_N, \ldots \rangle$ such that each frame $F_i \in \mathscr{F}$ is a set of IC3-lemma. Furthermore, the trace satisfies the following properties for $i \geq 0$:*

$$F_0 \equiv Init \tag{4.1}$$

$$F_i \wedge Tr \implies F'_{i+1} \tag{4.2}$$

$$F_i \implies F_{i+1} \tag{4.3}$$

$$i < N \implies (F_i \implies \neg Bad) \tag{4.4}$$

Intuitively, each frame $F_i \in \mathscr{F}$ over-approximates all the states reachable in at most i steps of the transition relation *Tr* from *Init*. The index i is the *level* of F_i. Moreover, the trace proves that $\mathscr{S}$ is safe up to $N-1$ steps of *Tr* from *Init*.

IC3 uses the trace to compute an increasing bound of steps from *Init* up to which $\mathscr{S}$ is safe. The algorithm works by iteratively adding an initially empty frame F_N at the end of the trace. IC3 then tries to either prove safety of F_N by strengthening it, or to find a feasible counterexample based on it.

Definition 6 (Proof Obligation) *Given an instance $\mathscr{S}$ of the safety problem and a* IC3 *trace $\mathscr{F}$ for $\mathscr{S}$, a* proof obligation *is the pair $\langle \sigma, i \rangle$ where σ is a conjunction of predicates over state variables and $i \leq N$. In addition, the proof obligation satisfies the following:*

- *$\sigma \wedge F_i$ is satisfiable, and*
- *for all models m such that $m \models \sigma$, $post^*_{Tr}(m) \wedge Bad$ is satisfiable.*

The conjunction σ represents a set of the states consistent with a frame $F_i \in \mathscr{F}$, containing states that can reach Bad with a feasible path.

Given an instance $\mathscr{S}$, IC3 computes a trace $\mathscr{F}$ of increasing length for $\mathscr{S}$ until either a fixed point is found for Tr or the algorithm determines a feasible counterexample. In the process, IC3 constructs candidate counterexamples, proof obligations, that are stored in an *obligation queue* $\mathscr{Q}$. The proof obligations $\langle \sigma, i \rangle$ are propagated towards the initial state by computing a formula σ^- such that $\sigma^- \wedge Tr \implies \sigma'$. The new proof obligation $\langle \sigma^-, i-1 \rangle$ is then inserted to $\mathscr{Q}$. If a counterexample candidate is not feasible, a proof obligation will at some point be blocked by a frame. This happens if for a proof obligation $\langle \sigma, i \rangle$ it holds that $F_{i-1} \wedge Tr \wedge \sigma'$ is unsatisfiable. A IC3-lemma φ, such that $\varphi \implies \neg\sigma$ is then inserted to F_i.

IC3 proves $\mathscr{S}$ safe if there exists $i < N$ such that $F_{i+1} \implies F_i$. This simplifies Equation (4.2) to $F_i \wedge Tr \implies F_i'$. Thus together with Equations (4.1) and (4.4) F_i is proved to be both a fixed point for Tr and a safe inductive invariant for $\mathscr{S}$. The way IC3 computes the fixed point leaves room for some flexibility in how the lemmas are organized. In particular [GI15] suggests to separate the inductive lemmas to a distinct frame F_∞. Hence the frame F_∞ is initially empty and always consists of those lemmas $\varphi \in \bigcup F_i$ inductive relative to F_∞. Thus, $\mathscr{S}$ is safe when $F_\infty \implies \neg Bad$.

IC3 proves $\mathscr{S}$ unsafe whenever a proof obligation $\langle \sigma, 0 \rangle$ is added to the obligations queue. By Definition 6, σ represents a set of states in *Init* (i.e. F_0) from which there is a feasible path leading to a state in *Bad*.

Definition 7 (IC3 Configurations) *Given an instance $\mathscr{S}$ of the safety problem, an* IC3 configuration *of $\mathscr{S}$ is the quadruple $\mathscr{C} = (N, \mathscr{F}, F_\infty, \mathscr{Q})$ where:*

- $N \in \mathbb{N}$,
- $\mathscr{F} = \langle F_0, \ldots, F_N, \ldots \rangle$ *is the* IC3 *trace of $\mathscr{S}$,*
- F_∞ *is the inductive frame,*
- $\mathscr{Q} = \{\langle \sigma, i \rangle, \ldots\}$ *is the obligation queue, where $i \leq N$.*

The Initial IC3 *configuration of $\mathscr{S}$ is $\mathscr{C}_0 = (1, \langle Init, \emptyset, \ldots \rangle, \emptyset, \emptyset)$*

Given a IC3 configuration $\mathscr{C}$ of a safety problem $\mathscr{S}$, each of the following rules performed on $\mathscr{C}$ updates its components resulting in a new configuration $\mathscr{C}'$. The components of $\mathscr{C}$ not mentioned in the rule are not updated by the rule application. The notation $F_{a \ldots b} \cup \{\varphi\}$ is equivalent to $F_i \cup \{\varphi\}$ for all $a \leq i \leq b$.

$$\textbf{Candidate: } \frac{\mathscr{Q}}{\mathscr{Q} \cup \{\langle \sigma, N \rangle\}} \quad \textbf{if } \begin{cases} \sigma \text{ is a formula} \\ \sigma \implies F_N \wedge Bad \end{cases}$$

$$\textbf{Predecessor: } \frac{\mathscr{Q}}{\mathscr{Q} \cup \{\langle \delta, i-1 \rangle\}} \quad \textbf{if } \begin{cases} \langle \sigma, i \rangle \in \mathscr{Q} \\ \delta \text{ is a formula, } m \text{ is a model} \\ \text{for all } m \models \delta, m \wedge Tr \models \sigma' \end{cases}$$

$$\textbf{Blocking: } \frac{\mathscr{Q} \cup \{\langle \sigma, i \rangle\} \mid \mathscr{F}}{\mathscr{Q} \mid F_{1\ldots i} \cup \{\varphi\}} \quad \textbf{if } \begin{cases} \textbf{Predecessor} \text{ is not applicable} \\ \varphi \text{ is a IC3-lemma} \\ Init \implies \varphi \\ \varphi \implies \neg\sigma \\ F_{i-1} \wedge Tr \implies \varphi' \end{cases}$$

$$\textbf{Unfold: } \frac{N}{N+1} \quad \textbf{if } \begin{cases} \textbf{Candidate} \text{ is not applicable} \\ \mathscr{Q} = \emptyset \end{cases}$$

$$\textbf{Inductive: } \frac{\mathscr{F} \mid F_\infty}{F_{1\ldots i} \cup \{\varphi\} \mid F_\infty \cup \{\varphi\}} \textbf{ if } \begin{cases} \varphi \text{ is a conjunction of IC3-lemmas} \\ \varphi \subseteq F_i, \text{ for some } 0 \leq i < N \\ \varphi \wedge F_\infty \wedge Tr \implies \varphi' \end{cases}$$

In addition, IC3 has the following two rules that guarantee the termination of the algorithm and are always taken when they are applicable:

$$\textbf{Safe: } \frac{}{Safe} \quad \textbf{if } \{ \; F_\infty \implies \neg Bad$$

$$\textbf{Unsafe: } \frac{}{Unsafe} \textbf{ if } \{ \; \langle \sigma, 0 \rangle \in \mathscr{Q}$$

In IC3 with theories, and, therefore, in this implementation, the operation *Predecessor* employs Model-Based Projection [KGC16] to ensure termination, while *Blocking* uses interpolation [HB12] to build the lemma.

Definition 8 (IC3 Strategy) *Given an instance $\mathscr{S}$ of the safety problem and a* IC3 *configuration $\mathscr{C}$, a* IC3 *strategy $\mathscr{T}_{\mathscr{S}}$ is a function that maps $\mathscr{C}$ to one of the possible* IC3 *rules applicable for $\mathscr{C}$ given $\mathscr{S}$.*

Given a IC3 strategy $\mathscr{T}_{\mathscr{S}}$, IC3 execution is a sequence of configurations $\langle \mathscr{C}_0, \mathscr{C}_1, \ldots, \mathscr{C}_T \rangle$ such that for every $i \in \{1, \ldots, T\}$, $\mathscr{C}_i$ is the result of the operation $\mathscr{T}_{\mathscr{S}}(\mathscr{C}_{i-1})$ on $\mathscr{C}_{i-1}$, $\mathscr{C}_0$ is the initial IC3 configuration for $\mathscr{S}$, i.e. $(1, \langle \mathit{Init}, \emptyset, \ldots \rangle, \emptyset, \emptyset)$, and $\mathscr{T}_{\mathscr{S}}(\mathscr{C}_T) \in \{\mathit{Safe}, \mathit{Unsafe}\}$.

4.2 The P3 Algorithm

This section introduces the P3 (Parallel IC3) algorithm for parallel model-checking with IC3. P3 implements three parallelisation techniques for IC3: *portfolio*, *partitioning*, and *lemma sharing*. These techniques can be combined in order to exploit the strengths of each other.

Portfolio uses different IC3 strategies while partitioning focuses the search by constraining the problem. In general, partitioning means dividing a problem into several sub-problems. The IC3 partitioning technique introduced in this section partitions the problem by restricting the paths leading to the bad states.

Finally, lemma sharing provides each solver with useful information arising from search diversification, possibly not derivable locally. The intuition is that IC3-lemmas express what is learned by each IC3 execution. Since different executions employ different strategies, IC3-lemmas that are easily found by one strategy can be difficult to find by another.

In the rest of the section, the concepts of portfolio, partitioning, and lemma sharing strategies are formalized providing details of the P3 algorithm.

4.2.1 Portfolio

The most straightforward parallel technique is a *portfolio* – concurrent and independent execution of multiple sequential IC3 strategies on the same problem instance. A portfolio terminates as soon as one of its instances terminates successfully.

The notion of a distributed IC3 configuration is defined to model a IC3 portfolio with any combination of lemma sharing and partitioning.

Definition 9 (Distributed IC3 Configuration) *Given an instance $\mathscr{S}$ of the safety problem, a distributed* IC3 *configuration of $\mathscr{S}$ is a set of tuples:*

$$\mathscr{D}^n = \{(\mathscr{T}^i_{\mathscr{S}}, \mathscr{C}^i)\}$$

where for each $i \in \{1,\ldots,n\}, n \in \mathbb{N}$:

- $\mathscr{T}^i_{\mathscr{S}}$ *is a* IC3 *strategy for* $\mathscr{S}$ *from Definition 8,*
- $\mathscr{C}^i = \left(N^i, \mathscr{F}^i = \langle F^i_0, F^i_1, \ldots, F^i_{N^i}, \ldots, \rangle, F^i_\infty, \mathscr{Q}^i\right)$ *is a* IC3 *configuration from Definition 7.*

A distributed IC3 configuration expresses a IC3 portfolio when for every $i \in \{1,\ldots,|\mathscr{D}|\}$, $\mathscr{T}^i$ is a strategy for the input problem $\mathscr{S}$. That is, every strategy executes the corresponding IC3 configuration independently, performing asynchronous and arbitrary choices.

A IC3 portfolio $\mathscr{D}$ terminates when there exists $\mathscr{T}^i_{\mathscr{S}}(\mathscr{C}^i) \in \{Safe, Unsafe\}$ for some $1 \leq i \leq |\mathscr{D}|$. Termination and soundness of this setting follows trivially from IC3 because every execution is independent.

4.2.2 Partitioning

This section defines partitioning strategy and argue for its soundness.

Definition 10 *Given a safety problem* $\mathscr{S}$*, partition*$(\mathscr{S})$ *is a set of instances of the safety problem* $\{\mathscr{S}_{p_1},\ldots,\mathscr{S}_{p_n}\}$*, where each instance* $\mathscr{S}_{p_i} = \langle Init(X), Tr(X,X'), p_i(X)\rangle$ *is called a partition of* $\mathscr{S}$*, and such that*

$$\bigvee_{i=1}^{n} p_i \iff \exists X' \cdot Tr(X,X') \wedge Bad(X')$$

From Definition 10, it follows that $\mathscr{S}$ is safe if and only if all of its partitions are safe. A distributed IC3 configuration $\mathscr{D}$ expresses partitioning if for each partition $\mathscr{S}_p \in partition(\mathscr{S})$ there is a pair $(\mathscr{T}^i_{\mathscr{S}_p}, \mathscr{C}^i) \in \mathscr{D}$. The result of a distributed configuration with partitioning is *Unsafe* if there exists $\mathscr{T}^i_{\mathscr{S}_p}(\mathscr{C}^i) = Unsafe$, and the result is *Safe* if for each $\mathscr{S}_p \in partition(\mathscr{S})$ there exist $\mathscr{T}^i_{\mathscr{S}_p}(\mathscr{C}^i) = Safe$.

The soundness is by construction: a counterexample for $\mathscr{S}_p$ is also valid for $\mathscr{S}$, while the safety of all $\mathscr{S}_p$ ensures the safety of $\mathscr{S}$ because every state leading to *Bad* in one step is expressed in a partition.

4.2.3 Lemma sharing

This section gives the formal definition of lemma sharing and argue for its soundness in a distributed portfolio setting.

Definition 11 (k-invariant) *A IC3-lemma ψ is* k-invariant *if it is true in all the states reachable in k steps or less, i.e., $post^k_{Tr}(Init) \implies \psi$. If a IC3-lemma φ is invariant, then it is k-invariant for any $k \in \mathbb{N}$.*

Following Definition 11, each frame F_k, $k \in \mathbb{N}$, is a set of k-invariants for $\mathscr{S}$, while F_∞ is a set of invariants for $\mathscr{S}$.

Theorem 1 (Lemma Sharing) *Given a distributed* IC3 *configuration $\mathscr{D}$ for an instance $\mathscr{S}$ of the safety problem, the IC3-lemma $\psi \in F^i_k, k \in \mathbb{N}$ is a k-invariant for $\mathscr{S}$ and the operation of adding ψ to any F^j_l with $i \neq j$ and $l \leq k$ keeps $\mathscr{C}^j$ a valid* IC3 *configuration for $\mathscr{S}$.*
The same holds for $\varphi \in F^i_\infty$ when added to any F^j_∞, $i \neq j$.

Proof. The proof follows from Definition 4. Each $\varphi \in F^i_k$ is a k-invariant if $k \in \mathbb{N}$, or an invariant if $k = \infty$ and can be used to soundly refine a different abstraction of states reachable in up to k steps. Similarly, sharing invariants is sound and makes every F^i_∞ an invariant for $\mathscr{S}$. Thus, $\mathscr{S}$ is safe whenever any F^i_∞ implies $\neg Bad$. □

4.2.4 Parallelly Performed IC3

The P3 algorithm is shown in Algorithm 1. P3 combines portfolio, lemma sharing, and partitioning. The algorithm works as follows. While there are available computing resources, the procedure **Entrust** at line 6 selects a partition not yet solved, creates a new strategy and allocates the necessary resources in order to run IC3. The procedure **Exclude** at line 12 is taken when the inductive frame of a IC3 configuration proves a partition unreachable. In this case, such partition is removed and all computing resources previously allocated are released. After **Exclude** is taken, all the computing resources available might be reallocated on other partitions by several **Entrust** calls. The procedure **Lemma Sharing** exchange IC3-lemmas between frames from different configuration having the same level. The procedures **Reachable** and **Unreachable** are taken respectively when a partition is proven reachable, and when all partitions are proven unreachable. When $partition(\mathscr{S}) = \{\mathscr{S}\}$ at line 1, then partitioning is disabled. When the procedure **Lemma Sharing** at line 5 is never taken then lemma sharing is disabled. If both are disabled then the algorithm corresponds to a IC3 portfolio.

Input : Safety problem $\mathscr{S} = \langle Init(X), Tr(X, X'), Bad(X)\rangle$.
Output : {*Reachable*, *Unreachable*}
Data : A distributed IC3 configuration $\mathscr{D}$, a set of partitions $\mathscr{P}$.
Initially: $\mathscr{D} \leftarrow \emptyset$, $\mathscr{P} \leftarrow \emptyset$.
Assume: $Init \wedge Bad$ is unsatisfiable.

1 $\mathscr{P} \leftarrow partition(\mathscr{S})$
2 **while** *True* **do**
3 **Reachable**: if $\langle \sigma, 0\rangle \in \mathscr{Q}^i$ for some $i \in \{1, \ldots, |\mathscr{D}|\}$, **return** *Reachable*.
4 **Unreachable**: if $\mathscr{P} = \emptyset$, **return** *Unreachable*.
5 **Lemma Sharing**: copy a IC3-lemma $\varphi \in F_n^i$ to F_n^j with: $i, j \in \{0, \ldots, |\mathscr{D}|\}$, $i \neq j$ and $n \in \mathbb{N} \cup \{\infty\}$
6 **Entrust**: if computing resources are available, then:
7 select a partition $\mathscr{S}_p \in \mathscr{P}$
8 create a new IC3 strategy $\mathscr{T}_{\mathscr{S}_p}$
9 create new $\mathscr{C} = (1, \langle Init, \emptyset, \ldots\rangle, \emptyset, \emptyset)$
10 set $\mathscr{D} \leftarrow \mathscr{D} \cup \{(\mathscr{T}_{\mathscr{S}_p}, \mathscr{C})\}$
11 allocate computing resources for IC3$(\mathscr{T}_{\mathscr{S}_p}, \mathscr{C})$
12 **Exclude**: if there exists $\mathscr{S}_p \in \mathscr{P}$ such that $F_\infty^i \implies \neg p$ for some $i \in \{1, \ldots, |\mathscr{D}|\}$, then:
13 $\mathscr{P} \leftarrow \mathscr{P} \setminus \{p\}$
14 release computing resources used for each $(\mathscr{T}_{\mathscr{S}_p}, \mathscr{C}) \in \mathscr{D}$
15 **end**

Algorithm 1: The P3 algorithm.

Soundness and Termination. Assuming $|\mathscr{P}|$ is finite, the procedure **Exclude** at line 12 is taken exactly $|\mathscr{P}|$ times, once for each $\mathscr{S}_p \in \mathscr{P}$. Therefore, the algorithm terminates if all IC3 instances executed by the procedure **Entrust** terminate. The only procedure that can affect termination of the individual IC3 executions is **Lemma Sharing**. Following theorem 1, the procedure **Lemma Sharing** provides each IC3 execution with valid IC3-lemmas that exclusively refine the frames, leading the execution toward convergence. In fact, the lemma φ makes the frame F_n^j a stronger conjunction. It is not possible for any frame to get weaker after **Lemma Sharing** is applied.

4.3 Experiments

This section presents an extensive experimental evaluation of the P3 algorithm on instances from the Software Verification Competition. The CPU time required

to run all the experiments is about 2775 CPU days, or 7.6 CPU years. The performance of the algorithm is measured separately on instances known to be easy and hard for the sequential model checker SPACER. The performance of combinations of different lemma sharing heuristics is studied both in a portfolio and by partitioning the input instance. Furthermore, this section compares different degree of parallelism to assess algorithm scalability by varying the number of solvers.

4.3.1 Experimental setup

All the reported experiments are run using SMTS [MHS18] in a cluster where each computing node is equipped with 20 CPU cores provided by 2×Intel E5-2650 v3 CPU, 64 GB of RAM and Intel 40Gbps Infiniband network adapter. For all the experiments, each computing node runs 10 solvers, and the server is executed in a separate node. In order to avoid memory management congestions, the number of solvers varies in an experiment by allocating a different number of nodes. The timeout is set to 1000 seconds wall-clock time.

The benchmark set used in this evaluations is constructed by SEAHORN [GKKN15], a fully automated analysis framework for the C language. Given as input the source file, SEAHORN constructs the triplet $\langle Init(X), Tr(X,X'), Bad(X)\rangle$ expressed in SMT-LIB v2 like language and representing the input safety problem instance. The benchmark set is based on 1,802 C problems taken from the SV-COMP 2016 Device Drivers Linux 64-bit (LDV) categories available at `https://github.com/sosy-lab/sv-benchmarks/tree/master/c` and preprocessed by SEAHORN.

First, the benchmarks are evaluated sequentially using the different strategies available in SPACER: IC3, GPDR and DEF. These settings are referred to as *sequential*. Those benchmarks solved in less than one second are removed from the set, experimenting over the remaining 562 benchmarks. Based on these evaluations, two different benchmarks sets of easy and hard instances are defined respectively.

- *less500*: benchmarks solved in less than 500 seconds by at least one strategy. In total 251 benchmarks.

- *more500*: benchmarks solved in more than 500 seconds or which timed out for at least one strategy. In total 325 benchmarks.

These benchmark sets partially overlap by having 14 benchmarks in common.

Table 4.1. Comparison between sequential (first three lines) and different parallel techniques using 60 solvers (following lines). The number of solved reachable, unreachable, and unknown instances are respectively labeled with #R, #UR, and #UK

Technique	*less500*			*more500*		
	#R	#UR	#UK	#R	#UR	#UK
SPACER(GPDR)	63	175	**13**	0	8	317
SPACER(IC3)	64	155	32	2	9	314
SPACER(DEF)	64	155	32	2	13	**310**
portfolio	66	185	**0**	8	40	277
∞-invariants	66	185	**0**	7	49	269
k-invariants	66	182	3	7	90	**228**
$*$-invariants	66	185	**0**	7	90	**228**
partitioning	66	176	9	10	34	281
partitioning+∞-invariants	66	183	2	11	49	265
partitioning+k-invariants	66	182	3	11	115	**199**
partitioning+$*$-invariants	66	185	**0**	16	98	211
virtual best	66	185	0	18	132	175

4.3.2 Comparing Parallel Techniques

This section presents a comparison among different combinations of parallel techniques in order to assess the effect of each technique features. For these experiments the number of solvers is fixed to 60, and portfolio and partitioning are combined with the 4 lemma sharing strategies: k-invariants, ∞-invariants, $*$-invariants, and none. The result is eight parallel techniques, each evaluated against both *less500* and *more500* benchmarks set.

Performance

Table 4.1 shows an overall evaluation for the experiments. For each technique, the table reports number of instances proven reachable, unreachable, and those unsolved, for both the benchmarks sets. The table is partitioned into 4 parts. Going from top to bottom: part 1 contains the results from sequential executions; part 2 contains the result for lemma sharing strategies with pure portfolio; part 3 contains results with partitioning; and part 4 presents the results of the *virtual best* solver that uses the best configuration for each problem. This *virtual best* is

Table 4.2. Average speedup compared to sequential solving.

Sequential strategy	*less500*		*more500*	
	60 CPU	virtual best	60 CPU	virtual best
SPACER(GPDR)	8×	10×	59×	91×
SPACER(IC3)	26×	32×	56×	88×
SPACER(DEF)	27×	33×	53×	83×

achievable by running in isolation a portfolio of the 8 combinations, therefore using 8×60 CPUs.

Notably, every parallel technique outperforms sequential execution. For the *more500* set the partitioning-based techniques perform the best. For the *less500* set, especially for reachable instances, portfolio-based technique is the best, matching the virtual best solver.

Table 4.2 reports average time speedups between sequential executions and the respective best parallel techniques for both sets, over benchmarks that did not time out. The columns *60 CPU* show the performance of the best technique: ∞-invariants for *less500* and partitioning+k-invariants for *more500*.

Figure 4.1 present the overall evaluation for the 8 parallel techniques considered, the 3 sequential strategies (IC3, GPDR and DEF), and the virtual best, each run on *less500* (left), and *more500* (right). Figure 4.1 (left) shows that sharing ∞-invariants over portfolio is the best technique for *less500*, already performing similarly to the virtual best. In fact, ∞-invariants solves all the benchmarks with an average 30% slowdown with respect to the virtual best, and up to 27× faster than sequential, as reported in Table 4.2.

Complementarity

Figure 4.1 (right) shows that considering *more500*, partitioning techniques are the best choices. In fact, the best technique for this set are partitioning+k-invariants, and partitioning+$*$-invariants. Notably, 40 benchmarks are solved by only one of these two techniques, making them complementary rather than one being strictly better than the other. Such complementarity is clearly visible in the scatter plot in Figure 4.2 (right) that compares their runtime for each instance in *more500*.

A possible way to address complementarity issues is by implementing a portfolio of multiple isolated parallel techniques, giving priority to the complemen-

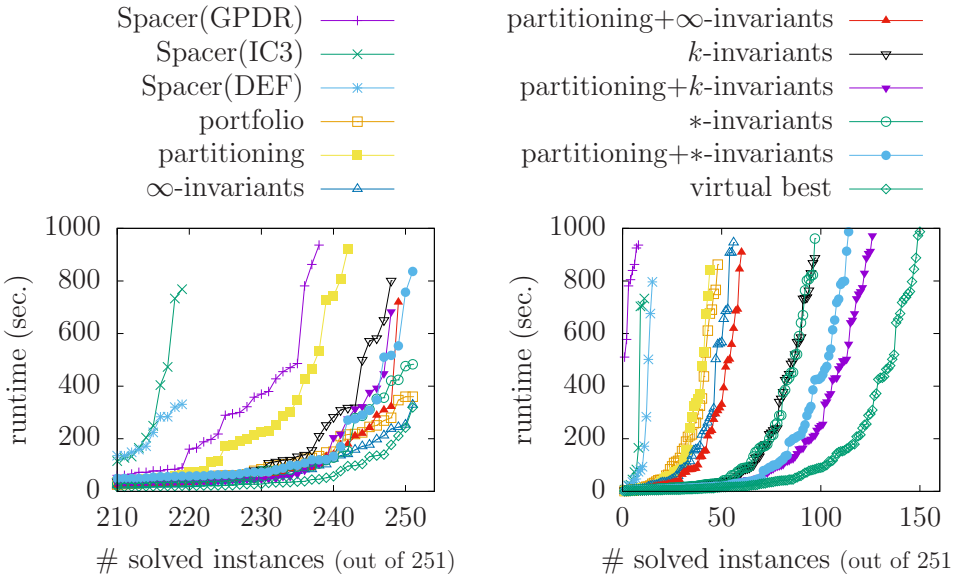


Figure 4.1. Comparison among all the considered techniques on the sets less500 (left) and more500 (right). k-invariants refers to sharing only lemmas from the trace, ∞-invariants refers to sharing just F_∞, while $*$-invariants implements both. Finally, for each benchmark the virtual best runtime among all the tested techniques is reported.

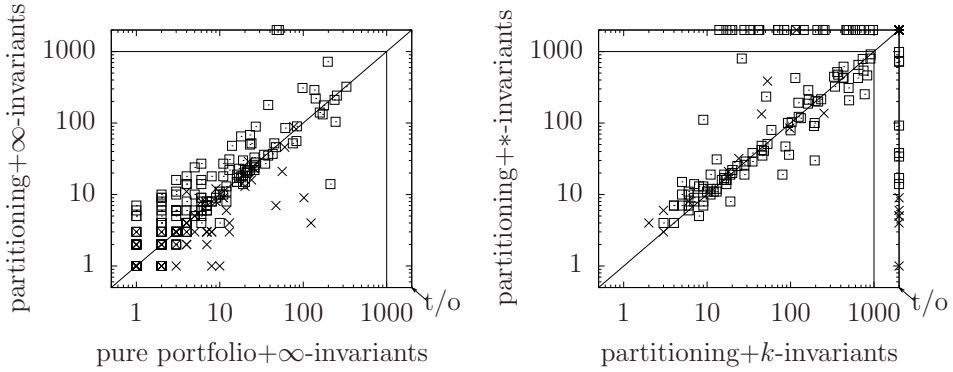


Figure 4.2. On the left: the comparison between sharing invariant with and without partitioning against less500. Compared to "best", this combination is overall 14% slower while using 75% less CPU. Moreover it is shown that partitioning outperforms on reachable instances. On the right: the comparison between sharing k-invariants and all lemmas with partitioning against more500. The two techniques are complementary, considering 40 instances are solved by only one of the two. The symbols $\times$ and $\boxdot$ respectively represent reachable and unreachable instances.

Table 4.3. Lemma sharing statistics.

Parallel technique	*less500* time	*less500* #lemmas	*more500* time	*more500* #lemmas
portfolio +				
∞-invariants	0.35%	141	0.41%	670
k-invariants	1.24%	252	1.00%	347
$*$-invariants	1.55%	243	0.83%	348
partitioning +				
∞-invariants	1.46%	170	0.87%	403
k-invariants	3.51%	140	4.55%	238
$*$-invariants	3.27%	221	4.45%	320

tary ones. This approach is capable of gradually improving performance toward the virtual best results, where the best performance is reached with the highest CPU resource allocation.

A portfolio of partitioning+k-invariants and partitioning+$*$-invariants for the set *more500* is capable of solving 140 instances, 10 less than the virtual best, and with an average 15% slowdown. Then, by adding $*$-invariants and k-invariants to such portfolio, it is possible to solve 148 instances with 5% slowdown and half computing resources with respect to virtual best. The missing 2 instances are only solved by partitioning+∞-invariants and ∞-invariants. Regarding *less500*, a similar setting can only increase CPU consumption without decreasing solving time. This is because ∞-invariants already solved all the benchmark. Figure 4.2 (left) shows the comparison between partitioning and portfolio with $*$-invariant sharing are quite complementary techniques. In fact, a portfolio of these two techniques is capable of solving the entire *less500* set using one fourth the CPU power and 14% slowdown compared to the virtual best. Another important result noticeable in Figure 4.2 (left) is that partitioning often outperforms pure portfolio on reachable instances. This is because the first partition proven satisfiable also proves the entire problem satisfiability, and the focused search done in each partition helps the solvers to converge quickly.

Lemma Sharing

Table 4.3 shows results about lemma sharing. The columns *time* show the average amount of time spent on lemma sharing push and pull, with respect to solving time. The columns *#lemmas* show the average number of IC3-lemmas exchanged.

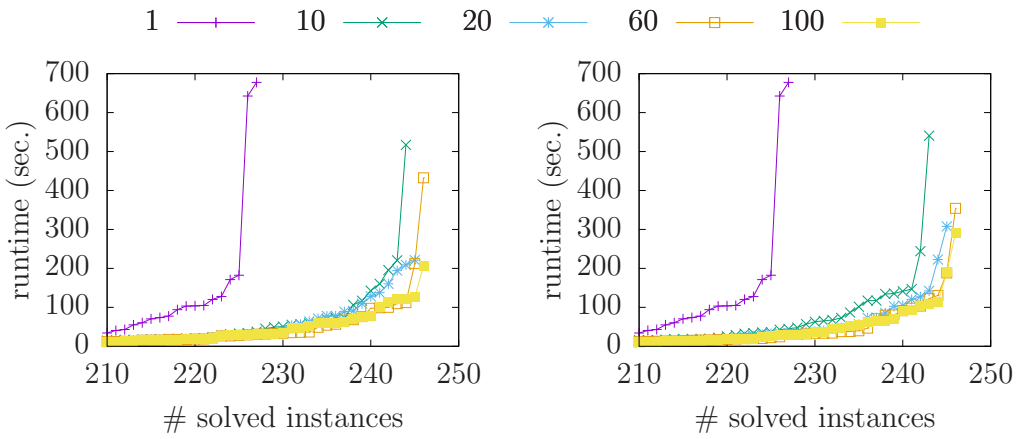
The amount of time spent on IC3-lemmas push and pull is about 1% and 3% of solving time, respectively, for portfolio based techniques and partitioning based technique. The average amount of IC3-lemmas generated is significantly lower than the amount of clauses produced in parallel SAT and SMT [HMS15, HJN09]. As a result, heuristics for limiting the amount of information exchanged to prevent considerable network delays are less crucial in the context of this work. Overall, the experimental evaluations show that parallel techniques are highly beneficial. In particular, parallelization with portfolio combined with sharing IC3-lemmas from F_∞ is the best choice for easier instances, while partitioning with sharing IC3-lemmas from the trace is the best choice for harder instances. This demonstrate that regardless the limited throughput, the choice of the lemma sharing strategy is important.

4.3.3 Scalability

This section evaluates scalability of the parallel techniques by experimenting different degrees of parallelism, i.e. by varying the number of solvers. Since each solver is a potentially expensive investment in computational power, it is of remarkable interest to assess the payoff of such investment in terms of solving time. The previous section evaluated every technique by setting the number of solvers to 60. For these experiments, the two best techniques for each benchmark set are considered and executed in isolation three more times: using 10, 20, and 100 solvers. In particular, the selected techniques are portfolio without lemma sharing and with ∞-invariants sharing for *less500*, and partitioning with k-invariants and $*$-invariants sharing for *more500*.

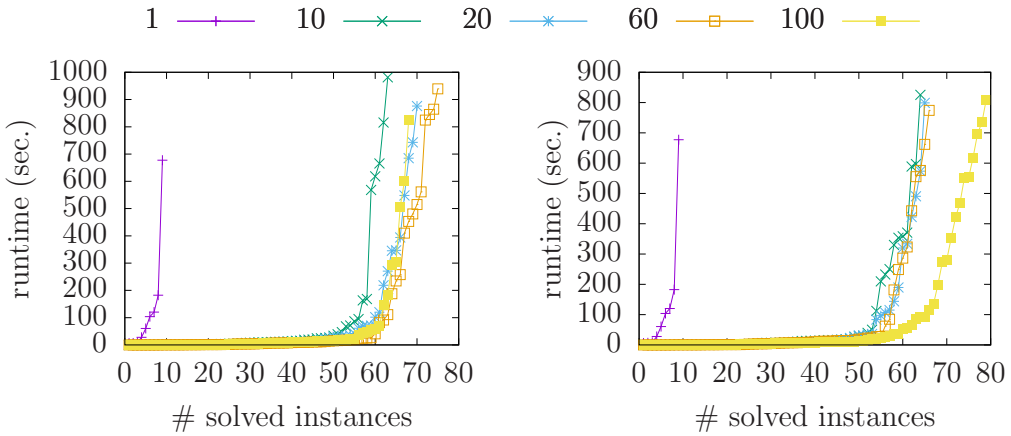
Figures 4.3 and 4.4 show, respectively for *less500* and *more500*, the comparison of different degrees of parallelism for executing the selected techniques. In both figures, the degree of parallelism '1' is the sequential execution of the IC3 strategy SPACER(GPDR), which is the one that solved most benchmarks (see Table 4.1). The tables at the bottom of both figures report, for each evaluated parallel technique, the speedup and the number of more instances solved with respect to using 10 solvers. The speedup considers the sum of the solving times of only those instances solved by all degrees of parallelism.

The techniques evaluated for *less500* appear to scale well based on Figure 4.3. In particular, increasing the number of solvers when sharing ∞-invariants leads to an initially higher speedup compared to portfolio. However, using 100 solvers such speedup recedes, making portfolio a better solution. This interesting observation witnesses that ∞-invariants are crucial for converging faster, however, too many solvers exchanging them can clutter the entire system and override



# Solvers	portfolio		∞-invariants	
	instances	speedup	instances	speedup
20	+1	+19%	+2	+23%
60	+2	+23%	+3	+32%
100	+2	+38%	+3	+36%

Figure 4.3. Comparison among different degrees of parallelism for portfolio without lemma sharing (left), and portfolio with ∞-invariants sharing (right) on less500. The table on the bottom reports speedup and more instances solved compared to 10 solvers.



Parallelism degree	partitioning+k-invariants		partitioning+$*$-invariants	
	instances	speedup	instances	speedup
20	+7	+33%	+1	+14%
60	+12	+53%	+2	-3%
100	+5	+31%	+15	+35%

Figure 4.4. Comparison among different degrees of parallelism for partitioning with k-invariants sharing (left), and partitioning with $*$-invariants sharing (right) on more500. The table on the bottom reports speedup and more instances solved compared to 10 solvers.

their positive effect. The same phenomenon is not observed for portfolio because there is no data exchanged during solving. As a result, it is statistically likely that by incrementing the number of solvers, the performance either stays unchanged or improves. However, such positive curve might hit a ceiling, meaning that no solving strategy is able to do better alone.

A substantially different scenario is presented in Figure 4.4. Partitioning+k-invariants performs very well considering the steep speedup increase reported up to 60 solvers. However, it suffers from scalability issues with 100 solvers. Conversely, partitioning+$*$-invariants scalability is initially worse for 60 solvers, but then improves and reaches the best performance with 100 solvers. In order to study better this experimental result, fig. 4.5 reports, for both techniques, a detailed comparison between the performances with 60 and 100 solvers performances over every benchmark. The chart in Figure 4.5 *left* shows that partitioning+k-invariants using 60 and 100 solvers are complementary, considering 21 benchmarks are solved by using either 60 or 100 solvers, not both. The chart on the right shows that 100 solvers clearly outperforms 60 for partitioning+$*$-invariants.

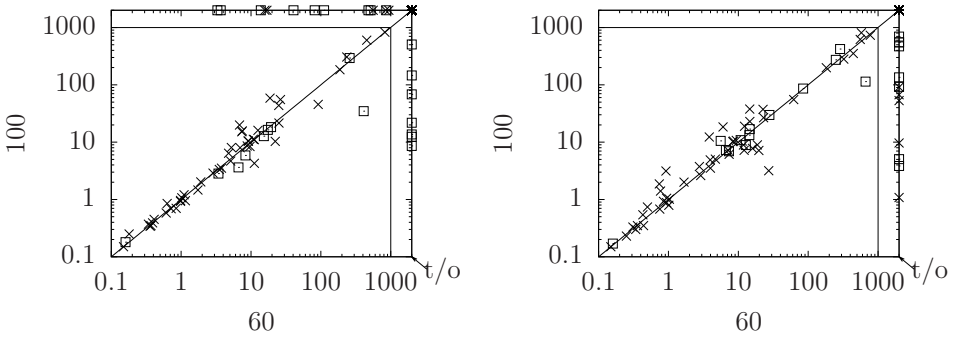


Figure 4.5. Comparison between degree of parallelism 60 and 100 on partitioning with k-invariants sharing (left), and partitioning with $*$-invariants sharing (right) on more500. The symbols × and ⊡ respectively represent reachable and unreachable instances.

This suggests us that lemmas, in particular ∞-invariants which are excluded by the latter technique, might play an important role in this opposed behaviour.

By considering only the subset of benchmarks solved by both techniques and by both degrees of parallelism, an interesting correlation arises between performance and lemmas exchanged per second. On average, the technique partitioning+$*$-invariants with 60 and 100 solvers, respectively produces 5.6 and 5.2 lemmas every second, thus decreasing lemmas by increasing solvers. Contrarily, the average production for partitioning+k-invariants are respectively 6.2 and 6.8 lemmas every second, therefore increasing lemmas by increasing solvers. This suggests us a possible future research direction on analysing lemmas impact on performance, considering also the overall lemma throughput of the system.

4.3.4 Comparison Against Sally

The parallel techniques performance are evaluated against the solver SALLY [JD16], the winner of the CHC-COMP 2019 for the category *transition systems* (TS). SALLY is designed for solving infinite-state transition systems, therefore the TS category is a perfect match. Contrarily, SPACER is a generic CHC solver, and was ranked third for TS. The comparison between parallel techniques using SPACER against sequential SALLY for the TS category is of particular interest. The motivation is to assess the trade-off between the available options to improve solving targeting a specific domain. In particular, assessing how convenient it would be to implement parallel techniques on top of an already existing solver, with respect to design a specific solver from scratch. Considering the effort required in

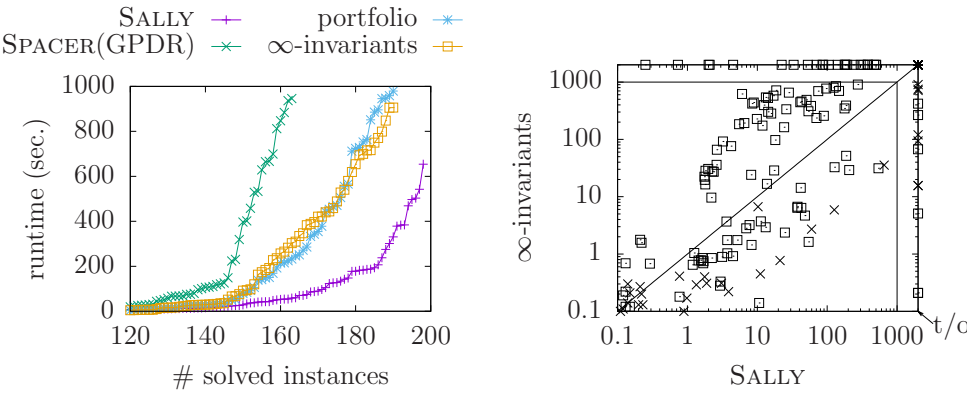


Figure 4.6. On the left, the comparison between sequential Spacer(GPDR) and Sally, and the parallel techniques portfolio and ∞-invariants using 60 solvers. On the right, the detailed comparison between sequential Sally and ∞-invariants.

implementing and maintaining a solver, the former option is easier to achieve. However, the latter would likely produce better results.

Figure 4.6 *left* shows a comparison between sequential SPACER, sequential SALLY, and the parallel techniques portfolio and ∞-invariants on top of SPACER using 60 solvers. Sequential SPACER solved 163 instances. Both parallel techniques solved 27 more benchmarks, 190 in total. Sequential SALLY solved 8 more instances than the parallel techniques, 198 in total, with an average speedup of 4.9× with respect to ∞-invariants. However, Figure 4.6 *right* shows that the variance of such speedup is very high, considering in many cases ∞-invariants performs extremely better by solving up to 100× faster and 23 more instances than sequential SALLY. Finally, sequential SALLY solved up to 10× faster and 31 more instances than ∞-invariants.

4.4 Related work

The first attempt to parallelize IC3 is mentioned in the original IC3 paper [Bra11], where the experimented parallel setting is based on sharing all the frames among different computing threads on the same machine. This work is further improved in [CK16] where they study different parallel approaches, all of them focused on multi-threaded portfolios and IC3-lemma sharing, addressing propositional IC3.

Both [Bra11, CK16] are limited to propositional IC3, making them suitable mainly for hardware verification. In contast to [Bra11, CK16], this work proposes and thoroughly evaluate different lemma sharing strategies by differentiating be-

tween k-invariants and ∞-invariants inside the frames. This chapter introduces the novel partitioning technique for IC3, together with a concise algorithm capable of combining all these techniques in a sound manner. Moreover this work's implementation is based on the more scalable distributed computing, thus exploiting the much bigger computational power offered by cloud-computing environments compared to single-machine threads.

A first investigation of IC3 in the setting of software verification is done in [CG12]. A different approach based on CHC for software verification is implemented in the tool SPACER [KGC16]. Individual improvements of SPACER algorithms are QUIP [GI15] and QUIC3 [GSV18]. QUIP brings to SPACER a more efficient way to produce system invariants and helps convergence. QUIC3 exploits the use of quantifiers in system invariants to progress the search faster. This chapter extends this track of work with the aim of improving the current state-of-the-art of software verification performed with parallel computation.

An algorithm combining IC3 and k-induction is presented in [JD16] and implemented in the tool SALLY. The effectiveness of this parallel techniques is compared with SALLY over infinite-state transition systems, where SALLY is shown empirically to perform particularly well by a recent software verification competition.

There is a substantial amount of work on parallel SMT solving that can help software verification model checking techniques in general. The parallelization tree framework for combining divide-and-conquer and portfolio directly on SMT formulas is introduced in [HMS15] and augmented with clause sharing in [MHS16]. A parallel approach for model checking of concurrent programs is given in [Hol16], while [RSMO15] presents a parallel symbolic execution involving several sequential SMT solvers.

4.5 Conclusions and Future Work

This chapter presented the parallel approach for IC3-inspired algorithms for software model checking published in [MGHS17]. The P3 algorithm is based on combining in a IC3-specific way diversification through algorithm portfolios, and cooperation through divide-and-conquer and exchange of information learned during the IC3 execution. The algorithm and its parallel extensions are described in a unified framework that allows both to reason about the correctness of the implementation and to study the effect of each component in relative isolation. The empirical evaluation is performed through an implementation of P3 on top of the solver SPACER, providing execution on both distributed environments and

multi-core through the SMTS [MHS18] framework. The experimental results obtained over a representative set of software verification benchmarks from the SV-COMP 2016 competition and CHC-COMP 2019 show that the parallel approach is vastly superior to sequential SPACER configurations, solving over hundred more instances within our timeout and providing super-linear speed-ups on average and good scalability when using up to one hundred CPUs. Furthermore, the comparison between our parallel implementation and SALLY over transition systems benchmarks witnesses that a general CHC solver, when parallelized, performs similarly to a solver designed for a particular kind of benchmarks.

In summary, the contributions of this chapter are: the definitions of IC3 partitioning; the P3 algorithm with soundness and termination proofs; the implementation using the sequential solver SPACER [KGC16] as a basis; the thorough experimental analysis processing 1802 instances from the software verification competition 2016 (SV-COMP) to evaluate each parallel technique, combination of techniques, and their scalability; the comparison agains the solver SALLY with instances from the constrained Horn clauses competition (CHC-COMP) 2019.

The contributions presented in this chapter open several research directions for the future. A promising future direction is to scale the system up to support thousands of solving agents. This direction introduces challenges on how to efficiently exchange data among the agents. Possible engineering solutions involve network broadcasting and multicasting. Alternatively, in order to limit the amount of data being exchanged, an interesting solution involves sharing a subset of the available IC3-lemmas to a subset of the available solving agents. This would be achieved by performing a selection in each solving agent before sending the IC3-lemmas over to the peer agents. How to perform such selection effectively is highly non-trivial and requires further research. Furthermore, IC3 partitioning can be performed iteratively. In particular, each solving agent can produce partitions of its entrusted partition. The task might be guided by heuristics that attempt to produce partitions similarly hard to solve. Both directions can be combined and performed at the same time, resulting in a substantial amount of research efforts needed to devise solutions that can be scaled up effectively.

Finally, the techniques for parallelizing SMT solving presented in chapter 3 offer an opportunity to investigate their synergy when applied in the IC3 context. For instance, SMT-specific information resulting from the individual solving tasks during IC3 executions might be beneficial to other solvers when processing similar SMT queries. Such information include theory lemmas that express valid statements under specific theories. When exchanging such lemmas, it is highly non-trivial to match the original statement (i.e. to express the same logical fact) in the receiving solver given its internal state. However, considering the com-

putational complexity of the theories involved, it is interesting to evaluate the effects of exchanging SMT-specific information between individual queries during IC3 executions.

Chapter 5

SMTS: multi-agent cooperative constraints solving

Mathematical formulations, i.e. constraints, are widely used for modelling systems in formal verification and optimization. The high complexity of the corresponding solving problem limits however the scalability of this approach to face more complex instances. Algorithm parallelization helps overcoming this limitation by exploiting parallel hardware architectures, or even better, distributed computing clusters. However, constraint techniques differ considerably between different applications, and often each needs a tailored parallelization procedure. An important challenge in constraint solving is therefore the design of parallel techniques capable of scaling to high degrees of parallelism, and extendible by domain-specialists to a wide range of different applications.

This chapter presents the tool SMT *Service* (SMTS), a framework for multi-agent cooperative solving offering a flexible API to support many different solving engines, specifically designed to relieve solvers developers from the burden of correctly handling concurrency and protocol details for agents communication. The contributions in this thesis on parallel solving show SMTS being general enough to achieve multi-agent *T*-DPLL-based SMT solving in chapter 3, and IC3-based CHC solving in chapter 4. This chapter focuses on the details of SMTS API and architecture and reports results from the related publication [BHMS20] witnessing SMTS usefulness to support parallelization of the class of PD-KIND algorithms [JD16].

The non-deterministic behaviour caused by the interleaving of sequential executions in different machines could make identifying correctness and performance problems an overwhelming task. To help understanding the complex parallel executions, SMTS offers a GUI that allows users to inspect at a high

level the executions both in real time and by browsing their history. In addition, the GUI provides the SMT-specific CNF visualization as a variable interaction graph [Sin07] enhanced with theory-specific features and learned clauses highlight.

The rest of this chapter is organized as follows. Section 5.1 presents details of the API and the GUI. Then, section 5.2 shows results from [BHMS20] on how SMTS is used for parallelizing PD-KIND and provides experimental data. Finally, sections 5.3 and 5.4 respectively presents related work and concludes the chapter.

5.1 SMTS Architecture

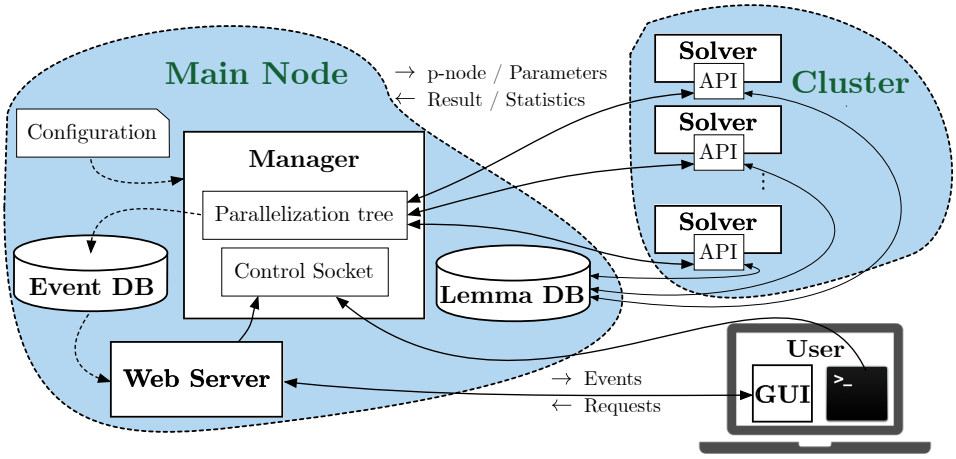


Figure 5.1. SMTS framework overview. Solid lines represent TCP/IP connections, while dashed lines represent disk I/O.

The SMTS architecture (fig. 5.1) consists of several components representing processes running on different computing nodes and communicating using TCP/IP. The *manager* receives tasks from the user through the *control socket*, which can be accessed either through the *terminal interface*, or through the GUI. A *configuration file* provides both general settings (e.g. parallelization tree and network configurations), and solver-specific parameters that will be forwarded together with each p-node solving task. The *Parallelization tree* keeps track of the mapping between *solvers* and p-nodes by distributing the instances of each unsolved p-node among all the available solvers. Events such as solver failures and additions occurring during the execution are managed soundly by the server. The API layer each solver implements makes the underlying algorithms transparent to the rest of the framework. The *lemma database* stores and provides lemmas to

the solvers, filtering lemmas based on different heuristics. The history of events related to the solving task is stored in the *event database* which can be inspected using the GUI either live or once the solving has terminated.

5.1.1 Application Program Interface

The API takes care of handling communication between the solvers and both the manager and lemma database. This includes scheduling incoming solving tasks, reporting solving results and statistics, and importing and exporting lemmas. The API consists of the methods of the `C++` class `solver`, that provide the SMTS functionalities to any given solver. The abstraction with the manager is provided by five `solver` class methods: `init()` for initializing the solver, `solve()` for solving a given instance, `partition()` for creating a given number of partitions of the current solving instance, `interrupt()` for interrupting the solver, and `report()` for reporting solving status and statistics.

The first four methods are only declared and must therefore be implemented using the application specific solving engine code. The abstraction for exchanging lemmas through the lemma database is provided by the two class methods `lemma_pull()` and `lemma_push()`. From the perspective of the SMTS implementation, a lemma is an smtlib formula associated with the parallelization tree node currently being solved by the solver. Lemmas complying with the smtlib format specification make the cooperation between different solvers natively possible. Each solver is responsible for proper lemma marshaling and unmarshaling, in particular for taking care of uniqueness and self-containedness.

SMTS API versatility is proven by implementing a parallel SMT solver based on OpenSMT2 [HMAS16], and a parallel IC3 solver based on Spacer [KGC16]. Both OpenSMT2 and Spacer implementations share basic design principles: `init()` initializes the required solver's data structures by setting parameters as requested by the manager (contained in the configuration file); `solve()` provides the given instance to the underlying engine, and calls the solving function whose output is then given as input to `report()`. Using just these SMTS features, a portfolio of solvers can be easily obtained by properly initializing solver randomness in `init()`.

5.1.2 Graphical User Interface

One of the most prominent features of SMTS is its ability to visualize and guide the problem solving. This section gives details on the functionalities provided by the SMTS GUI.

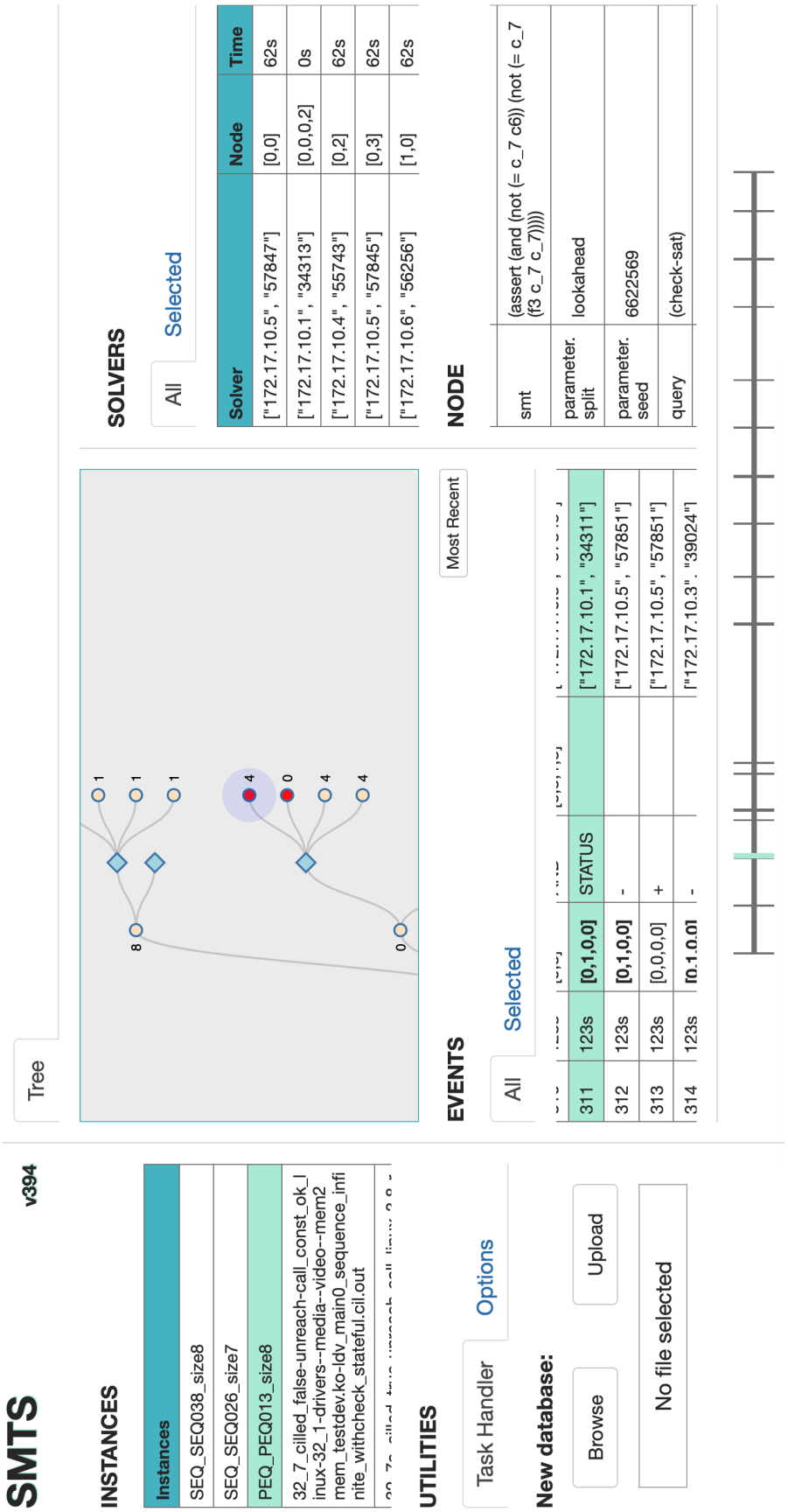


Figure 5.2. SMTS GUI visualizing an unsafe SMT instance.

The web server acts as a key component in the visualization, gathering and sending to the client web application information on the solving-related events by querying an event database. If configured for live mode, the GUI is also connected to the manager control socket, enabling the user to interact with the current solving tasks.

The graphical user interface consists of two views: the *main view* showing the parallelization tree, and the *instance view*, available only during live solving. We describe both the views in this section.

The data relevant to the parallelization is received from the web server and displayed through six views, allowing the user to analyze and guide the execution of SMTS. This includes in particular retrieving statistics, parameters and past events of interest, but also initiating partitioning at will. Figure 5.2 shows the SMT Viewer client web application with the six views. We provide a brief explanation for each of the views.

Utilities. In the live mode, this view shows the name of the instance currently being solved, and the time left until the solving times out. The user can upload new instances, change the timeout or terminate solving. In the off-line mode, the user can upload and analyze different databases.

Instances. This view contains a list of all the instances available in the connected event database. After selecting a particular instance, all other views will refer only to the selected instance. Instances are ordered by scheduling time with the one at the bottom being the most recent.

Events. The view is composed of two interactive components: a table showing all the events related to the selected instance, and a time-line displaying how the events are distributed over time. A feature we found useful in this view is that when the user selects either an event or a point in the time-line, all the other views are updated to reflect the status at that time. This allows the user to "rewind" the solving execution and better analyze the interesting events occurred during solving. Browsing the past is the typical way of finding performance problems and other anomalies. The *selected* tab allows the user to filter events, showing only those related to the node currently selected in the tree view.

Tree. The parallelization tree is presented here as it was at the time of the selected event. Each p-node is associated with an integer indicating the number of solvers working on the associated instance. A single click on a node results in all

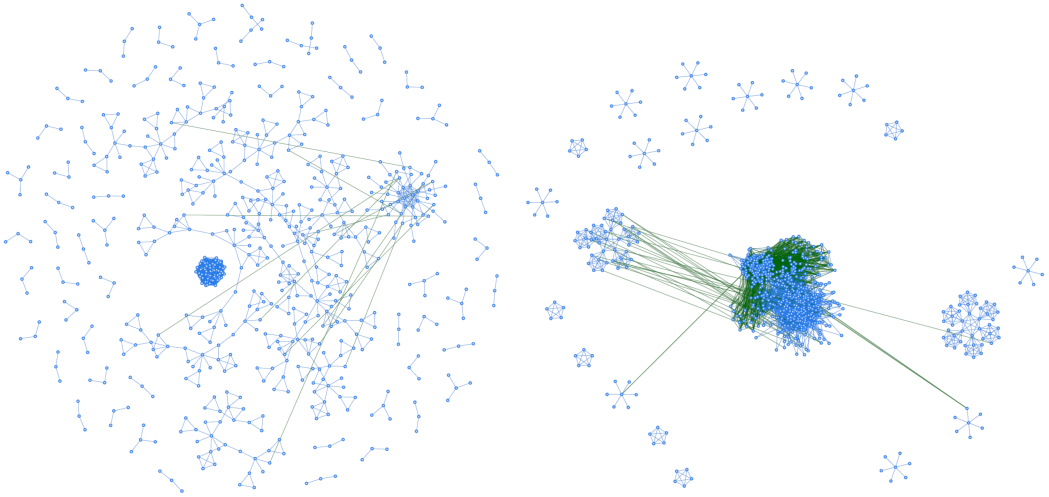


Figure 5.3. SMTS GUI CNF visualization example on two different QF_LRA instances. Green edges represents learned binary clauses.

the node-specific information being updated in the node view. The colour of the nodes changes according to the type (p-nodes or r-nodes) and status (satisfiable, unsatisfiable or unknown)

This view has special features in live mode. A double click on a p-node currently being solved triggers live partitioning of such node. If the tree belongs to an instance currently being solved, the *CNF* tab becomes available and enables displaying the CNF structure of the node together with the learned binary clauses. Two examples of CNF visualization are given in fig. 5.3. Further details about this visualization are given in the next section.

Solvers. This view shows each solver together with its assigned node at the selected event time. The user can then analyze how many solvers were present and to which node they were assigned at each phase of the solving process. The *selected* tab allows the user to show only the solvers working on the currently selected node in the tree view.

Node. This view reveals information about all the data related to the selected tree node. If the node is solved, the view shows the statistics and the parameters of the solver who solved the instance associated with the node.

5.1.3 SMT Formula Visualization

The CNF visualization interface provides a variable interaction graph [Sin07] based on the CNF structure of the instance, enhanced with SMT-specific visualization features. The basic graph is drawn based on the CNF structure. However, the interactive interface supports also visualization of the connections imposed by the theory structure. An example of the SMT instance visualization is given in Figure 5.3. In the GUI, clicking on a node the tool shows a menu listing the theory variables appearing in the node. By enabling theory variables and the Boolean selector function *and* or *or*, one can highlight with a colour the nodes that contain either all or one of the selected theory variables.

The graph is drawn using the `javascript` library `vis.js`[1] which computes the placements for the nodes based on a physical model. The tool also supports interactively moving the nodes, which we found useful in inspecting the connections of a node on a highly connected cluster.

Currently the CNF visualization feature provides support for viewing learnt binary clauses as separate edges in the graph. We also experimented with visualizing longer clauses. This resulted in our examples an overly connected graph, and is therefore disabled in the released version.

One could think of alternative ways of representing the SMT instances which would more closely show the SMT structure. However, we argue that the one based on variable interaction graph is relevant for the SMT solvers since they rely heavily on SAT instances in the solving process. We also experimented with visualizing the factor graph [Sin07] of the instance. Even though SMT instances are typically simpler in the CNF structure than the corresponding SAT instances, the factor graph was excessively heavy for visual rendering for our instances.

5.2 Multi-agent IcE/FiRE

In multi-agent cooperative IcE/FiRE (presented in details in [BHMS20]), several solving agents work on the same problem and exchange information. The communication is divided between the finite reachability engines and the induction-checking engines. An instance of the IcE/FiRE framework is shown in fig. 2.1.

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,

[1] `https://visjs.org`

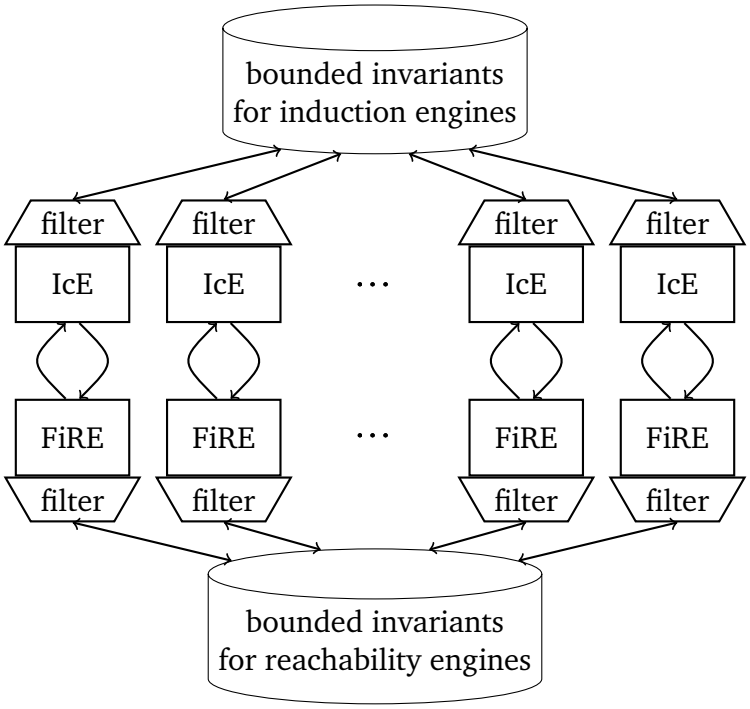


Figure 5.4. Multiple instances of IcE/FiRE framework sharing information

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. 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 $\mathscr{G}$. When an engine is working on a level n and a lemma is pushed to $\mathscr{G}$, 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.

Parallel PD-KIND

Since PD-KIND is an instantiation of the IcE/FiRE framework (see details in [BHMS20, JD16]), it can be readily plugged into the abstract parallel framework with information sharing described in section 2.3.

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

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 usefulness of the lemma. Since PD-KIND assumes that each element of the base frame is associated with a potential counter-example through the mapping *CEX*, each bounded invariant l that is sent out by IcE must also be accompanied by its companion $\mathit{CEX}(l)$.

It is important for the success of a parallel approach to *diversify* the search for the solution. Here are listed the key points for PD-KIND diversification.

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 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 $\mathscr{F}^k$-inductive, and thus successfully pushed, but not

$\mathscr{F}^{k'}$-inductive for $k' < k$.

Obligation processing strategy. Several rules might be applicable given a configuration with 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 $CEX(l)$. Therefore, the behaviour of the algorithm can be controlled through the strategy determining the obligation to pick from the queue.

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 for computation of bounded invariants. However, Craig interpolant for a given problem is in general not unique and there exist techniques for computing different interpolants in propositional logic and in theories of first-order logic. The use of different interpolation algorithms leads to different bounded invariants and this can have a huge influence on the performance of the whole algorithm.

5.2.1 Experiments

The implementation of multi-agent cooperative PD-KIND algorithm is based on the open-source model checker SALLY [JD16] and uses the SMTS framework for parallelization and information exchange. SALLY is extended with APIs for sending and receiving information through SMTS. For the experiments of this section, SALLY uses YICES [Dut14] for checking satisfiability and OPENSMT2 [HMAS16] for the interpolation queries.[2]

The benchmarks are taken from the transition systems category of CHC COMP 2019[3], where the problem is encoded using the theory of linear real arithmetic. Out of 244 benchmarks, 7 problematic ones were excluded due to reasons such as the presence of a non-linear operations. All experiments were run on a single multi-core machine with 16 Intel® Xeon® X5687 @ 3.6 GHz CPUs and 180 GB

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

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

of RAM. The resources were restricted to 1000 seconds of timeout and 6GB of memory for each SALLY solving agent.

All solving agents use the default strategy of SALLY when they are choosing the depth of induction. The obligation processing strategy is a priority queue based on a score assigned to obligations, randomized to diversify the behaviour of different agents. The learning strategy is diversified primarily by using different interpolation algorithms in OPENSMT2 and secondary by using different random seed for the SMT search. Three different LRA interpolation algorithms were used: Farkas interpolation algorithm [McM05], dual Farkas, and an interpolation algorithm based on decomposing Farkas interpolants [BHKS19], respectively denoted as PF, DF and PD.

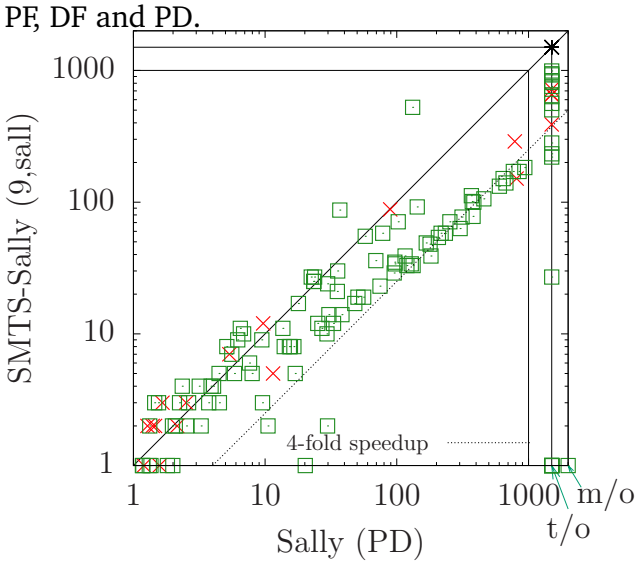


Figure 5.5. Best parallel configuration against the winner of LRA-TS category of CHC COMP 2019

Comparison to the state-of-the-art. The result of the experiments is shown in fig. 5.5 that compares the performance of the winner of the transition systems category of CHC COMP 2019 (sequential SALLY using PD interpolation algorithm in OPENSMT2) against the multi-agent SMTS-based SALLY with nine agents exchanging information between IcEs and between FiREs (9,sall). The parallel implementation achieves 4-fold speedup on a significant number of instances and solves 224 instances compared to 197 instances solved by the sequential version.

Cooperation by sharing information. Figure 5.6 summarizes the performance of 4 configurations using six agents: no information sharing (6,sno), sharing between FiREs only (6,sreach), sharing between IcEs only (6,sind), and all sharing

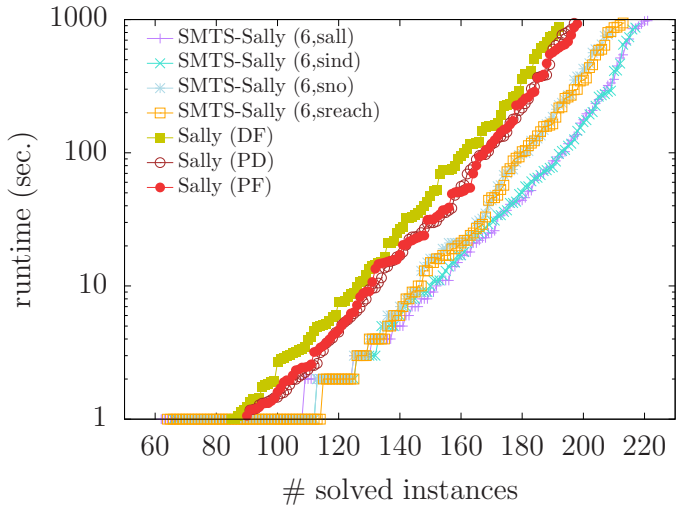


Figure 5.6. The effect of sharing information

enabled (6,sall). In all these configurations six agents were running, 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 the whole seconds and this creates an effect of "stairs" for the low runtimes in cactus plots with logarithmic scale. There is also a significant number of instances solved almost instantly and 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 (sno, sind) do not profit much after enabling it (sreach, sall). However, some hard benchmarks could only be solved by allowing reachability information to be shared. On the other hand, enabling the sharing of induction information does boost the performance significantly. We conclude that the best performance was achieved by enabling sharing information between both IcEs and FiREs.

Scalability. We compared the performance of one, two, six and nine agents with all information sharing enabled. For the experiment with two agents the interpolation algorithms used are PF and PD. The results, summarized in Fig. 5.7,

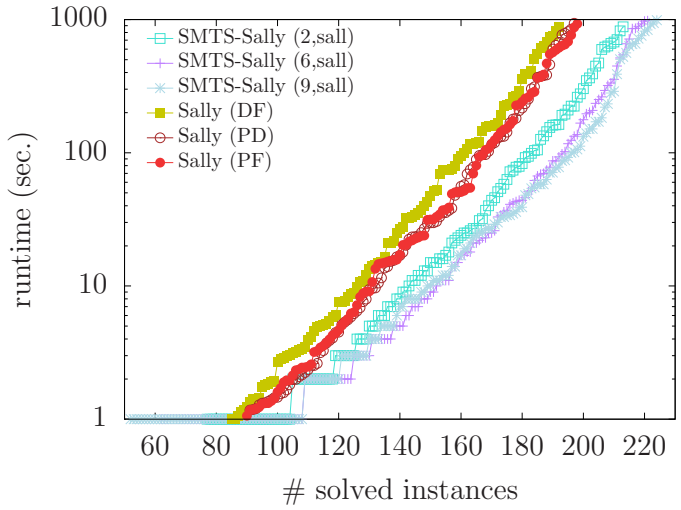


Figure 5.7. Scalability experiments

show that increasing the number of solving agents improves the performance, both decreasing the runtime and solving more benchmarks. The number of instances solved is 197, 213, 221 and 224, respectively for one, two, six and nine agents.

Agents diversification. The large jump when moving from sequential solving to two agents running in parallel can be in part contributed to different interpolation algorithms. We investigate this further in Figure 5.8. We compared configurations using six agents with diverse interpolation algorithms (6,sall) and with fixed interpolation algorithms PF and PD, respectively (6,sall,PF) and (6,sall,PF). We also added the configuration (2,sall) of just two agents, one using PF, and one using PD. The results show that varying the interpolation algorithm is very important as the performance of (2,sall) is comparable to that of (6,sall,PD) and (6,sall,PF). On the other hand, (6,sall) performs significantly better.

The experiments show that multi-agent SALLY performs substantially better than the sequential version. Its success can be contributed to more than one factor: The use of diversification by means of different interpolation algorithms helps to solve more benchmarks compared to a single interpolation algorithm. Cooperation by sharing information between solving agents can significantly reduce the runtime and thus solve more instances within the time limit. The major part of this can be contributed to the sharing of induction information, but sharing reachability information does help as well. The scalability experiments show continuing improvement up to nine solving agents.

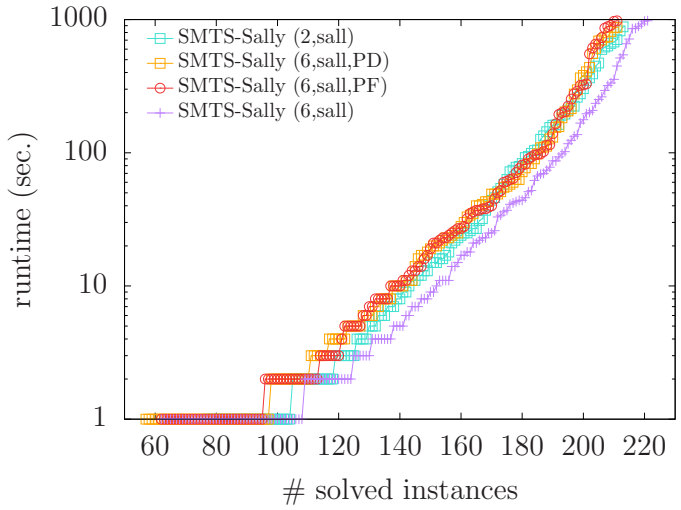


Figure 5.8. The effect of using different interpolation algorithms

5.3 Related work

The seminal paper on the model-checking algorithm IC3 [Bra11] already provides experimental evidence on the efficiency of parallelization. Parallelization of IC3 is further investigated in [CK16], and by the authors in [MGHS17] in the context of the parallelization tree formalism. Visualization of parallel executions of constraint logic programs is studied in [CH00]. Similarly to SMTS, the tool allows inspecting the execution at different time points, however without using the general parallelization tree formalism. Finally, our tool contributes to visualizing the structure of constraint problems and the executions of related, sequential search algorithms. In this domain it is of particular interest the work [Sin07] for SAT solving, [KS13] for answer set programming, and [KT11] for parallel k-induction.

5.4 Conclusions and Future Work

This chapter presented SMTS, the tool introduced in [MHS18], and the results obtained using SMTS in the related work [BHMS20]. In particular, this chapter presents the APIs for parallelizing a given sequential solving engine, discusses the features offered by the built-in interactive graphical user interface (GUI), and reports results of the usage of SMTS technology for parallelizing PD-KIND [BHMS20]. SMTS targets in particular clusters and cloud computing environments, which easily offer the computational power of hundreds of CPUs.

The SMTS web-based GUI allows users to visualize resource allocation, interactively guide problem solving by manually controlling the cluster resources usage, and study the history of events and statistics from individual solving tasks.

In summary, the contributions of this chapter are: the SMTS APIs for parallelization of different sequential solving engines; the web-based SMTS GUI for interaction and inspection of parallel executions, and SMT formulas visualization; the results of SMTS used to parallelize PD-KIND that confirm the APIs flexibility and usefulness.

Future directions in this context centre around improving the scalability and the GUI. Scaling SMTS to support thousands of agents entails the design of efficient techniques of message passing that comprise load balancing schemes. The overall performance would still depend of the underlying solving engines and their efforts to limit the amount of data to exchange. However, SMTS APIs could provide clever structures to ease the task of limiting data exchange and assist the solving engines with a feedback of the system performance in order to encourage adaptation. The GIU can be enhanced with features specific to lemma sharing. In fact, it currently lacks of any visualization for the data being exchanged by the agents.

Chapter 6

Accurate Smart Contract Verification through Direct Modelling

Smart contracts challenge the existing, highly efficient techniques applied in symbolic model checking of software by their unique traits not present in standard programming models. Still, the majority of reported smart contract verification projects either reuse off-the-shelf model checking tools resulting in inefficient and even unsound models, or apply generic solutions that typically require highly-trained human intervention.

This chapter presents a solution focused on the accurate modelling of the central aspects of Solidity smart contracts based on their control-flow, enabling fully automatic verification. In this context, the term accurate refers to the property of the model to encode the semantic traits specific for smart contracts precisely, as opposed to replace them with non-deterministic operations. Successfully solving the model results in two possible outcomes. A finite-length counterexample consisting of a concrete list of transactions that produce a property violation, or a *contract invariant* that proves unbounded safety by expressing conditions over the contract variables that always hold after any possible transactions. The model is based on constrained Horn clauses (CHCs) [BGMR15]. Therefore, the multi-agent techniques presented in chapter 4 can be directly used to handle the solving task, allowing smart contract verification to easily take advantage from the scalability benefits offered by parallelization.

Besides the core contribution of the CHC modelling for smart contracts, this chapter reports the results from the collaboration with Leonardo Alt – formal verification engineer at Ethereum Foundation, and Rodrigo Otoni – colleague PhD student at USI Lugano. In particular, the collaboration resulted in the implementation of the CHC modelling technique inside the formal engine of the Solidity

compiler [Eth18b] developed by the Ethereum Foundation, and in the extensive experimentations and comparative analysis against related tools over thousands of recently-deployed smart contracts. Solidity [sol20] is the most popular language for developing ETHEREUM smart contracts [Eth18a].

The rest of this chapter is organized as follows. Section 6.1 introduces the concept and notation of CHCs. Sections 6.2 and 6.3 respectively present the modelling algorithm in details and provide an end-to-end example. Sections 6.4 and 6.5 respectively present details of the implementation and report experimental data over real-world contracts. Section 6.6 focuses on related work and section 6.7 concludes the chapter.

6.1 Background

In [BG87] the *Existential Positive Least Fixed-Point logic* (E+LFP) is proven to logically match Hoare logic [Hoa69] and is therefore useful for determining partial correctness of programs. Following [BGMR15], this chapter uses a specialization of E+LFP called *constrained Horn clauses* (CHC) due to the intuitive syntax in representing transition systems with loops, and the efficient decision procedures available for them. A characterisation of CHC is given based on first-order logic and the fixed-point operator adapted from [BG87]. Let ψ be a first-order formula over a theory T with free variables $\vec{x}$, and a finite set $\{P_1, \ldots, P_n\}$ be predicates over $\vec{x}$ not appearing in ψ. The satisfiability of $\psi(\vec{x}) \wedge P_1(\vec{x}) \wedge \ldots \wedge P_n(\vec{x})$ in theory T when the interpretations of P_i are Δ_{P_i} is denoted by

$$\bigcup_{i=1}^{n} \{\Delta_{P_i}\} \models_T \psi(\vec{x}) \wedge P_1(\vec{x}) \wedge \ldots \wedge P_n(\vec{x}).$$

Given a set of predicates $\mathscr{P}$, a first-order theory T, and a set of variables $\mathscr{V}$, a *system of CHCs* is a set S of clauses of form

$$H(\vec{x}) \leftarrow \exists \vec{y}.\, \phi(\vec{x}, \vec{y}) \wedge P_1(\vec{y}) \wedge \ldots \wedge P_m(\vec{y}) \text{ for } m \geq 0 \tag{6.1}$$

where ϕ is a first-order formula over $\vec{x}, \vec{y} \subseteq \mathscr{V}$ with respect to the theory T; $\vec{x}$ is the tuple of distinct variables free in ϕ; $H \in \mathscr{P}$ a predicate with arity matching $\vec{x}$; $P_i \in \mathscr{P}$ predicates with arities matching $\vec{y}$; and no predicate in $\mathscr{P}$ appears in ϕ. For a clause c, $head(c) = H$ and $body(c) = \exists \vec{y}.\, \phi(\vec{x}, \vec{y}) \wedge P_1(\vec{y}) \wedge \ldots \wedge P_m(\vec{y})$.

For each predicate $P \in \mathscr{P}$ the transfinite sequence Δ_P^α is given by

$$\begin{array}{l}\Delta_P^0 = \emptyset \\ \Delta_P^{\alpha+1} = \Delta_P^\alpha \cup \{\vec{a} \mid \bigcup_{Q \in \mathscr{P}} \{\Delta_Q^\alpha\} \models_T \bigvee_{c \in S, head(c)=P} body(c)[\vec{a}/\vec{x}]\} \\ \Delta_P^\lambda = \bigcup_{\alpha<\lambda} \Delta_P^\alpha \text{ for limit ordinals } \lambda.\end{array}$$

Since the sequence Δ_P^α is monotonic, there is a value for α such that $\Delta_P^\alpha = \Delta_P^{\alpha+1} = \Delta_P$.

In the context of modelling and verification, this chapter focuses of determining whether the $\Delta_\bot$ of the predicate $\bot \in \mathscr{P}$ is empty. In particular the CHC solver used in this chapter guarantees that if $\Delta_\bot$ is nonempty then the model of a program violates a safety property and the solver is able to map the construction to an execution. Conversely, if $\Delta_\bot$ is empty, the solver either does not terminate, or provides quantifier-free first-order formulas $\psi_P(\vec{x})$ in T for each $P \in \mathscr{P}$ that serve as safe inductive invariants in the following sense. First, each ψ_P over-approximate the interpretations Δ_P, that is,

$$\{\Delta_P\} \models_T P(\vec{x}) \implies \psi_P(\vec{x}).$$

Second, for each clause $c \in S$ of the form (6.1) where $head(c) \neq \bot$,

$$\models_T \phi(\vec{x}, \vec{y}) \wedge \psi_{P_1}(\vec{y}) \wedge \ldots \wedge \psi_{P_m}(\vec{y}) \implies \psi_H(\vec{x}).$$

Third, if $head(c) = \bot$, then

$$\models_T \neg\big(\phi(\vec{x}, \vec{y}) \wedge \psi_{P_1}(\vec{y}) \wedge \ldots \wedge \psi_{P_m}(\vec{y})\big).$$

The terminology from [BGMR15] is followed. A set of CHCs is *satisfiable* if $\Delta_\bot$ is empty, and *unsatisfiable* otherwise.

In presenting the clauses, some conventions are omitted in order to make reading them easier. First, the existential quantifier is omitted since its scope is clear from the arguments of the body for a given clause. Second, variables that that do not appear in the formulas are not written. Third, superfluous equalities are omitted: if an element y_i of $\vec{y}$ is equated with an element x_j of $\vec{x}$ in a top-level conjunct of ϕ, the equality is not written and substituted y_i for x_j in the head.

6.2 The Model

We define a contract C with the triplet $\langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$, where $\boldsymbol{s}$ is the set of state variables, $I(\boldsymbol{s})$ is the initial state of $\boldsymbol{s}$, and F is the set of all functions in the contract. The disjoint subsets F^+ and F^- of F denote respectively the sets of external and internal functions of F. Given a function $f(\boldsymbol{a}) \to \boldsymbol{r} \in F$, where $\boldsymbol{a}$ is the set of function arguments and $\boldsymbol{r}$ is the set of return variables, the control-flow graph (CFG) of f is the tuple $\langle G, \alpha, \omega, \rho \rangle$. $G = (V, E, \lambda, \mu, S)$ is a node- and edge-labeled directed graph, where V is the set of CFG blocks; $E \subseteq V \times V$ is the set of control flow *jumps*; λ_v is the set that contains, for all $v \in V$, the set of instructions performed by v; μ_e is, for all $e \in E$, the condition under which the jump e is performed; and $S \subseteq V$ is the set of *safety blocks*, each representing a safety property. During the execution of f only local variables are manipulated. Therefore the labelings λ and μ, respectively, of each block and jump, are instructions performed only over a set of local variables $\boldsymbol{l}$ of f. The CFG blocks $\alpha, \omega \in V$ are respectively the entry block and the exit block. The injection $\rho : \boldsymbol{s} \cup \boldsymbol{a} \cup \boldsymbol{r} \to \boldsymbol{l}$ maps every state variable, function argument and return variable to a distinct local variable accessed by the instructions in each block and jump. The function notation is extended to sets in the natural way: for a given set of variables $\vec{z}$, $\rho(\vec{z}) = \{\rho(x) \mid x \in \vec{z}\}$.

A safety property in the CFG is represented by a safety block. In Solidity, safety properties are specified with the `assert` keyword. Safety properties failing during the execution cause the function to revert and return immediately. To achieve this behaviour, for every safety block $b \in S$ there exists the jump $e = \langle b, \omega \rangle$ where the condition μ_e is the negation of the property. This ensures a direct jump to the exit block in case the safety property is violated. A jump to the exit block ω from a safety block requires ω to revert by restoring the state prior the function's execution. In order to provide ω with the information that a safety property has been broken, λ_b sets the special variable $\tilde{r} \in \boldsymbol{l}$ to a value that uniquely identifies the violated safety property.

Consider functions f and f' (which can be the same), represented by CFGs G and G' respectively. Function calls are performed by a block v in G whose labeling λ_v contains the call instruction to G'. At runtime, the execution of the CFG block v is performed by executing the CFG block α of G'. When ω of G' is executed, the transaction represented by the execution of G' is finalized by committing any changes to the state variables. The execution is then resumed from v, mapping the return variables of f' to the expected local variables of f, and updating the local variables of f representing state variables to match the new values resulting from the commit just performed by the concluded transaction.

6.2.1 Model of a Contract Function

This section presents the rules for creating the CHC model of a function $f(\boldsymbol{a})$ of a contract having state variables $\boldsymbol{s}$, returning variables $\boldsymbol{r}$, and manipulating local variables $\boldsymbol{l}$.

The CHCs are constructed given the control flow graph $\langle G, \alpha, \omega, \rho \rangle$ of the function f, where $G = (V, E, \lambda, \mu)$. For each CFG block v, the *Static Single Assignment (SSA) formula* $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$, where $\boldsymbol{l}' = \{x' \mid x \in \boldsymbol{l}\}$, models the behavior of v by formalizing in logic the relation between x and x' for each $x \in \boldsymbol{l}$, based on the execution of the instructions in λ_v. The formula $\mathrm{SSA}_{\mu_e}(\boldsymbol{l})$ of each jump e is the logical condition under which e is taken. For each CFG block $v \in V$, $\mathscr{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l})$ is a predicate symbol representing the states that are reachable in the block v. The set of rules representing the execution of f is defined as follows. For each jump $e = \langle v, u \rangle \in E$, the *jump rule* of e is the CHC

$$\mathscr{P}_f^u(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}') \leftarrow \mathscr{P}_f^v(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \mathrm{SSA}_{\mu_e}(\boldsymbol{l}). \qquad (\mathsf{Jump}_{f,e})$$

The *entry rule* sets the local variables equal to the corresponding current values of state variables and passed arguments.

$$\mathscr{P}_f^\alpha(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \leftarrow \bigwedge_{x \in \boldsymbol{s} \cup \boldsymbol{a}} x = \rho(x) \wedge \rho(\tilde{r}) = 0. \qquad (\mathsf{Entry}_f)$$

The variables in $\boldsymbol{s}$ and $\boldsymbol{a}$ are symbolically assigned in (Entry_f) and never changed throughout the jump rules $(\mathsf{Jump}_{f,e})$ of any $e \in E$. In case of reverting during execution, these variables provide the necessary information to revert to the state prior to the execution of f. A revert is caused a jump to ω setting the local variable $\rho(\tilde{r})$ equal to the integer identifier of a safety property that failed. Initially, $\rho(\tilde{r})$ is set to zero. Let $\mathscr{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r})$ be the predicate symbol representing the *function summary* of the execution of f. The function summary expresses the relation between the input and the output of an execution of the function. In this context the input is represented by the function arguments $\boldsymbol{a}$ and state variables $\boldsymbol{s}$ prior execution, and the output is represented by the return values $\boldsymbol{r}$ and the state variables $\boldsymbol{s}'$ after the execution. The *summary rule* of f is the CHC

$$\mathscr{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \leftarrow \mathscr{P}_f^\omega(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \qquad (\mathsf{Sum}_f)$$
$$\underbrace{\Big(\rho(\tilde{r}) \neq 0 \implies \bigwedge_{x \in \boldsymbol{s}} x' = x\Big)}_{\textit{revert}} \wedge \underbrace{\Big(\rho(\tilde{r}) = 0 \implies \bigwedge_{x \in \boldsymbol{s}} x' = \rho(x)\Big)}_{\textit{commit}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{r}} x = \rho(x)}_{\textit{returns}}.$$

The *revert* constraints in (Sum_f) ensures that an execution is reverted when ω

is reached having the local variable corresponding to $\tilde{r}$ set to the identifier of a safety property. Conversely, the mutually exclusive *commit* constraints store the local copy of the state in $\boldsymbol{s}'$, modeling a commit of the computed values. The *return* constraints equate the return variables $\boldsymbol{r}$ with the corresponding local variables.

Definition 12 *Given a contract function f, the set of CHC Π_f modeling f is the set consisting of the jump rule of e ($\mathsf{Jump}_{f,e}$) for each control flow jump e of f, and the entry and summary rules from f (Entry_f) and (Sum_f).*

6.2.2 Function Calls

Let $e = \langle v, u \rangle$ be a control flow jump where λ_v contains a function call to $g(\boldsymbol{a}_g)$ returning variables $\boldsymbol{r}_g$. The summary of g is used to synchronize the local variables of f with the new state committed after g's execution terminates. Therefore, $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is defined as

$$\mathscr{S}_g(\boldsymbol{s}', \boldsymbol{a}_g, \boldsymbol{s}'', \boldsymbol{r}_g) \wedge \qquad (\mathsf{Call}_{g,\rho_{call}})$$
$$\underbrace{\bigwedge_{x \in \boldsymbol{a}_g \cup \boldsymbol{r}_g} x = \rho_{call}(x)}_{\text{arguments and returns passing}} \quad \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)'\right)}_{\text{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\text{untouched locals}}$$

where $\rho_{call} : \boldsymbol{a}_g \to \boldsymbol{l}, \boldsymbol{r}_g \to \boldsymbol{l}'$ is the mapping specific for this call that maps both arguments of g to $\boldsymbol{l}$ according to how they are passed, and the return variables of g to $\boldsymbol{l}'$ according to how they are assigned; $\boldsymbol{l}_{call} = \rho_{call}(\boldsymbol{r}_g) \cup \rho(\boldsymbol{s})$ is the set of local variables that can be affected by the call. Arguments are assumed to be passed by value. Therefore local variables $\rho_{call}(\boldsymbol{a}_g)$ corresponding to the arguments of g are not affected by the execution of the block. The *argument and return passing* uses ρ_{call} to match arguments and return variables to the respective local variables of the caller. The *state set and update* conjunction makes sure that the local variables in $\boldsymbol{l}'$ representing the state variables get updated according to the execution of the just-ended transaction. For each local variable not in $\boldsymbol{l}_{call}$, the *untouched locals* constraint equates its primed and non-primed versions, modeling that its value is not affected by the block execution, and therefore remains unchanged after the jump. Note that the primed version of the local variables in $\boldsymbol{l}_{call}$ are set in the former constraints according to the effects of the call. This ensures that all variables in $\boldsymbol{l}'$, which are passed to the predicate $\mathscr{P}_f^u$, are constrained, modeling a deterministic execution. By applying ($\mathsf{Jump}_{f,e}$), the resulting CHC is non-linear because it contains the two predicates $\mathscr{P}_f^v$ and $\mathscr{S}_g$.

6.2.3 Contract's External Behaviour

Given a contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$, a contract transaction is the execution of a public function. A single contract transaction is therefore modelled by the summaries of every function f in F^+, each proving the relation between state variables $\boldsymbol{s}, \boldsymbol{s}'$ before and after a transaction performed by calling f. The *external behaviour* of the contract is defined as the transitive closure of contract transactions, modelling an arbitrary number of calls to any public function, in any order. The external behaviour provides the relation between state variables before and after any possible interaction with the contract performed by an external contract.

The predicate $\mathscr{E}_C(\boldsymbol{s}, \boldsymbol{s}')$ is defined to model the external behavior of C inductively, where the base case is the CHC

$$\mathscr{E}_C(\boldsymbol{s}, \boldsymbol{s}) \leftarrow \top, \tag{ExtBase$_{\mathscr{C}}$}$$

and the inductive steps are, for each function f in F^+, the CHCs

$$\mathscr{E}_C(\boldsymbol{s}, \boldsymbol{s}'') \leftarrow \mathscr{E}_C(\boldsymbol{s}, \boldsymbol{s}') \wedge \mathscr{S}_f(\boldsymbol{s}', \boldsymbol{a}, \boldsymbol{s}'', \boldsymbol{r}). \tag{ExtInd$_{\mathscr{C},f}$}$$

The external behaviour of C can be used to model calls to a function of an external contracts D which source code in unknown before runtime. In this way, any possible transaction resulting from D interaction during runtime is considered. Every control flow jump $\langle v, u \rangle$ in C, where the block v contains a call to a function that is unknown before runtime, is modelled using $\mathscr{E}_C$ in place of the called function summary. The resulting $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is built similarly to (Call$_{g,\rho_{call}}$), with the difference of omitting the *argument and return passing* constraints. The local variables in ρ_{call} are unconstrained in order to nondeterministically model any possible values returned by the unknown function. Specifically, the resulting $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is

$$\mathscr{E}_C(\boldsymbol{s}', \boldsymbol{s}'') \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)' \right)}_{\textit{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\textit{untouched locals}}. \tag{ECall$_{\rho_{call}}$}$$

If a safety proof for this model can be obtained, then it is not possible to construct an external contract that can violate assertions in C by any sequence of reentrant calls. A counterexample for such model implies that there exists a contract that can be designed specifically for violating one or more assertions, by calling one or more public functions in a particular order and returning specific values.

Input : A contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$.
Output : The set of CHC Π_C.
Initially: $\Pi_C = \{(\mathsf{Init}_{\mathscr{C}}), (\mathsf{ExtBase}_{\mathscr{C}})\}$.

foreach $f = \langle G, \alpha, \omega, \epsilon, \rho \rangle \in F$ **do**
  Let $\boldsymbol{a}, \boldsymbol{r}, \boldsymbol{l}$ respectively the arguments, returns and local variables of f.
  Let $\Pi_f := \{(\mathsf{Entry}_f), (\mathsf{Sum}_f)\}$
  Let $G = (V, E, \lambda, \mu)$
  foreach $e = \langle v, w \rangle \in E$ **do**
    if *v contains a call to* $g(\boldsymbol{a}_g) \to \boldsymbol{r}_g$ **then**
      Create ρ_{call} from λ_v
      if (Sum_g) *is known* **then** $\mathrm{SSA}_{\lambda_v} := (\mathsf{Call}_{g,\rho_{call}})$;
      else $\mathrm{SSA}_{\lambda_v} := (\mathsf{ECall}_{\rho_{call}})$;
    else
      $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') := \mathsf{Model}(\lambda_v)$
    end
    $\mathrm{SSA}_{\mu_e} := \mathsf{Model}(\mu_e)$
    $\Pi_f := \Pi_f \cup \{(\mathsf{Jump}_{f,e})\}$
  end
  $\Pi_C := \Pi_C \cup \Pi_f$
  if $f \in F^+$ **then**
    $\Pi_C := \Pi_C \cup \{(\mathsf{ExtInd}_{\mathscr{C},f}), (\mathsf{RootTr}_{\mathscr{C},f})\}$
  end
end

Algorithm 2: The algorithm to construct Π_C.

6.2.4 Checking Contract Safety

Let $\mathscr{C}(\boldsymbol{s})$ be the predicate representing the reachable values for the contract. The initial state is modeled by the CHC

$$\mathscr{C}(\boldsymbol{s}) \leftarrow I(\boldsymbol{s}). \qquad (\mathsf{Init}_{\mathscr{C}})$$

Every transition performed by a call to a public function is modeled by the *root transition rule*. For each public function $f \in F^+$,

$$\mathscr{C}(\boldsymbol{s}') \leftarrow \mathscr{C}(\boldsymbol{s}) \wedge \mathscr{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \wedge \tilde{r} = 0. \qquad (\mathsf{RootTr}_{\mathscr{C},f})$$

Definition 13 *Given a contract C, the set of CHC Π_C modeling any possible behavior of C is defined as the union of the initial rule $(\mathsf{Init}_{\mathscr{C}})$, the external base case rule $(\mathsf{ExtBase}_{\mathscr{C}})$, all the rules Π_f of every function $f \in F$, and for each public function $f \in F^+$ the root transition rule $(\mathsf{RootTr}_{\mathscr{C},f})$ and the external inductive rule*

$(\mathsf{ExtInd}_{\mathscr{C},f})$.

Algorithm 2 gives an overall view of the modeling technique. Given as input a smart contract C, the algorithm returns the set Π_C of CHCs modeling C. Initially, Π_C consists only of the initial rule of C. Then, the loop from line 1 to 20 iterates over each contract function f, gradually producing the respective set Π_f that is finally merged with Π_C in line 16. The internal loop from line 5 to 15 iterates over every edge $\langle v, w\rangle$ of the CFG of f. The case where v is a block representing a function call is handled in lines 6 to 9, using either the summary of the called function or the external predicate. Otherwise, a formal model representing the block execution is generated in line 10, and used in the jump rule.

Definition 14 (Safety Rule) *The safety rule Σ_f for the CHC model of a public function f is $\bot \leftarrow \mathscr{C}(\boldsymbol{s}) \wedge \mathscr{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \wedge \tilde{r} \neq 0$. The safety rule of a contract C is the set Σ_C of the safety rules of every public function of C.*

The safety rule ensures that a function f is safe, in the sense that every possible transaction of f does not revert, i.e. produce assertion violations. A contract C is safe if and only if the set $\Pi_C \cup \Sigma_C$ is satisfiable.

6.2.5 Counterexample Generation

The *refutation*, or proof of unsatisfiability, for $\Pi_C \cup \Sigma_C$ proves that a specific safety query in Σ_C can not be satisfied, i.e., $\Delta_\bot$ is non-empty. While the presented solving methodology can show satisfiability over unbounded executions through the use of over-approximation, it can only represent finite counterexamples. This, of course, is not a practical limitation since in real programs the interest is in bugs that manifest themselves after a finite number of steps. While the description of how a counter-example is constructed in the solver is outside of the scope of this chapter, this section gives a short overview of the refutations themselves.

A refutation is a tree-shaped structure obtained by an unwinding of clauses. The nodes of the refutation are labeled with clauses. The root v_0 of the tree is labeled with a clause with $\bot$ as head. For each predicate P in the body of a clause c, we create a child labeled with a unique clause c' such that $head(c') = P$. The leaves of the tree are labeled with clauses with no predicates in the body. Let $v_0, \ldots, v_k$ be a path from the root to a leaf, labeled with clauses $c_0, \ldots, c_k$. Given a clause c of form (6.1), let $body_\phi(c)$ denote the constraint ϕ of c. Then in a refutation for all such paths it must hold that

$$\models_T body_\phi(c_0)(\vec{x_0}, \vec{x_1}) \wedge body_\phi(c_1)(\vec{x_1}, \vec{x_2}) \wedge \ldots \wedge body_\phi(c_k)(\vec{x_{k-1}}, \vec{x_k}). \quad (6.2)$$

```
1  contract Auction {
2      uint bid = 0;
3      uint cash = 0;
4      address payable winner = address(0);
5
6      function offer() public payable {
7          uint new_bid = msg.value - 5 finney;
8          require(bid < new_bid);
9          if (winner != address(0)){
10             assert(bid <= cash);
11             winner.transfer(bid);
12             cash = cash - bid;
13         }
14         bid = new_bid;
15         cash = cash + msg.value;
16         winner = msg.sender;
17     }
18 }
```

Figure 6.1. The auction contract used as example.

A counterexample corresponds then to a first-order structure satisfying (6.2) as follows: The counterexample generation traverses the entire refutation tree and considers only the nodes that refer to the initial state rule ($\mathsf{Init}_{\mathscr{C}}$), the root transaction rule ($\mathsf{RootTr}_{\mathscr{C},f}$), or the safety rule. The breath-first search results in a list of nodes that has the safety rule as first element (the root), a possibly empty list of elements representing root transaction rules, and finally a leaf representing an initial rule. The first-order structure satisfying eq. (6.2) is used to produce a model of the initial state for the counterexample setup. Then, each following node represents the result of a transaction whose children model (i) the contract state prior the transaction, and (ii) a function call with given arguments that results in a new state. The last transaction involves a call to the function $\hat{f}$ that resulted in a revert. The arguments of each such function are then used to produce a trace of function calls which serves as the counterexample.

6.3 Example

This section considers the contract shown in fig. 6.1. The contract `Auction` provides a realistic support for an auction, where the function `offer` is used to place an offer by the users. Although in reality the contract would have other func-

$$\mathscr{P}_o^{\alpha} \leftarrow b = l_b \wedge c = l_c \wedge w = l_w \wedge s = l_s \wedge v = l_v \wedge l_{\tilde{r}} = 0 \qquad (\mathsf{Entry}_o)$$

$$\mathscr{P}_o^{9} \leftarrow \mathscr{P}_o^{\alpha} \wedge l_{nb} = l_v - 5 \times 10^{15} \wedge l_b < l_{nb} \qquad (\mathsf{Jump}_{o,\langle\alpha,9\rangle})$$

$$\mathscr{P}_o^{10} \leftarrow \mathscr{P}_o^{9} \wedge l_w \neq 0 \qquad (\mathsf{Jump}_{o,\langle 9,10\rangle})$$

$$\mathscr{P}_o^{14} \leftarrow \mathscr{P}_o^{9} \wedge \neg(l_w \neq 0) \qquad (\mathsf{Jump}_{o,\langle 9,14\rangle})$$

$$\mathscr{P}_o^{\omega} \leftarrow \mathscr{P}_o^{10} \wedge \neg(l_b \leq l_c) \wedge l'_{\tilde{r}} = 1 \qquad (\mathsf{Jump}_{o,\langle 10,\omega\rangle})$$

$$\mathscr{P}_o^{14} \leftarrow \mathscr{P}_o^{10} \wedge l_b \leq l_c \wedge l'_c = l_c - l_b \qquad (\mathsf{Jump}_{o,\langle 10,14\rangle})$$

$$\mathscr{P}_o^{\omega} \leftarrow \mathscr{P}_o^{14} \wedge l'_b = l_{nb} \wedge l'_c = l_c + l_v \wedge l'_w = l_s \qquad (\mathsf{Jump}_{o,\langle 14,\omega\rangle})$$

$$\mathscr{S}_o \leftarrow \mathscr{P}_o^{\omega} \wedge \underbrace{l_{\tilde{r}} \neq 0 \implies (b' = b \wedge c' = c \wedge w' = w)}_{\textit{revert}} \wedge$$

$$\underbrace{\neg l_{\tilde{r}} = 0 \implies (b' = l_b \wedge c' = l_c \wedge w' = l_w)}_{\textit{commit}} \wedge \underbrace{\tilde{r} = l_{\tilde{r}}}_{\textit{returns}} \qquad (\mathsf{Sum}_o)$$

$$\mathscr{A} \leftarrow b = 0 \wedge c = 0 \wedge w = 0 \qquad (\mathsf{Init}_{\mathscr{A}})$$

$$\mathscr{A} \leftarrow \mathscr{A} \wedge \mathscr{S}_o \wedge \tilde{r} = 0 \qquad (\mathsf{RootTr}_{\mathscr{A},o})$$

Figure 6.2. The set of CHCs Π_A that models the contract `Auction` shown in fig. 6.1.

tions for implementing additional functionalities (e.g. auction end, payment to the seller, ect.), for simplicity and to avoid unnecessary burden the focus is on providing a model and a counterexample only for the function `offer`. The contract state consists of three variables, `bid`, `cash`, and `winner`, which respectively represent the current winning bid, the amount of money held by the contract, and the address that made the current winning bid. Every new offerer pays a fee of 5 finney, (0.005 Ether, or 5×10^{15} wei) that is worth approximately 1 USD at the time of writing. The fee is deducted upfront from the amount sent by user when submitting the transaction (`msg.value`) on line 7, causing an underflow if such amount is less than the fee. As a result of the underflow, the current bid can potentially become a very large value, preventing other users to participate to the action and causing a denial of service. The assertion on line 10 checks that the contract has enough money to pay back a previous bidder if overcome by an higher offer. In case a previous transaction caused an underflow, the bid evaluated to a very large value and it is likely that the contract does not have enough money to cover the refund, causing the assertion to fail.

The set Π_A modelling the contract `Auction` is shown if fig. 6.2. The signature of the predicates is intentionally not present in order to avoid cluttering the notation. The signatures of the predicates $\mathscr{P}_o^{\alpha}, \mathscr{P}_o^{9}, \mathscr{P}_o^{10}, \mathscr{P}_o^{14}$ and $\mathscr{P}_o^{\omega}$ represent-

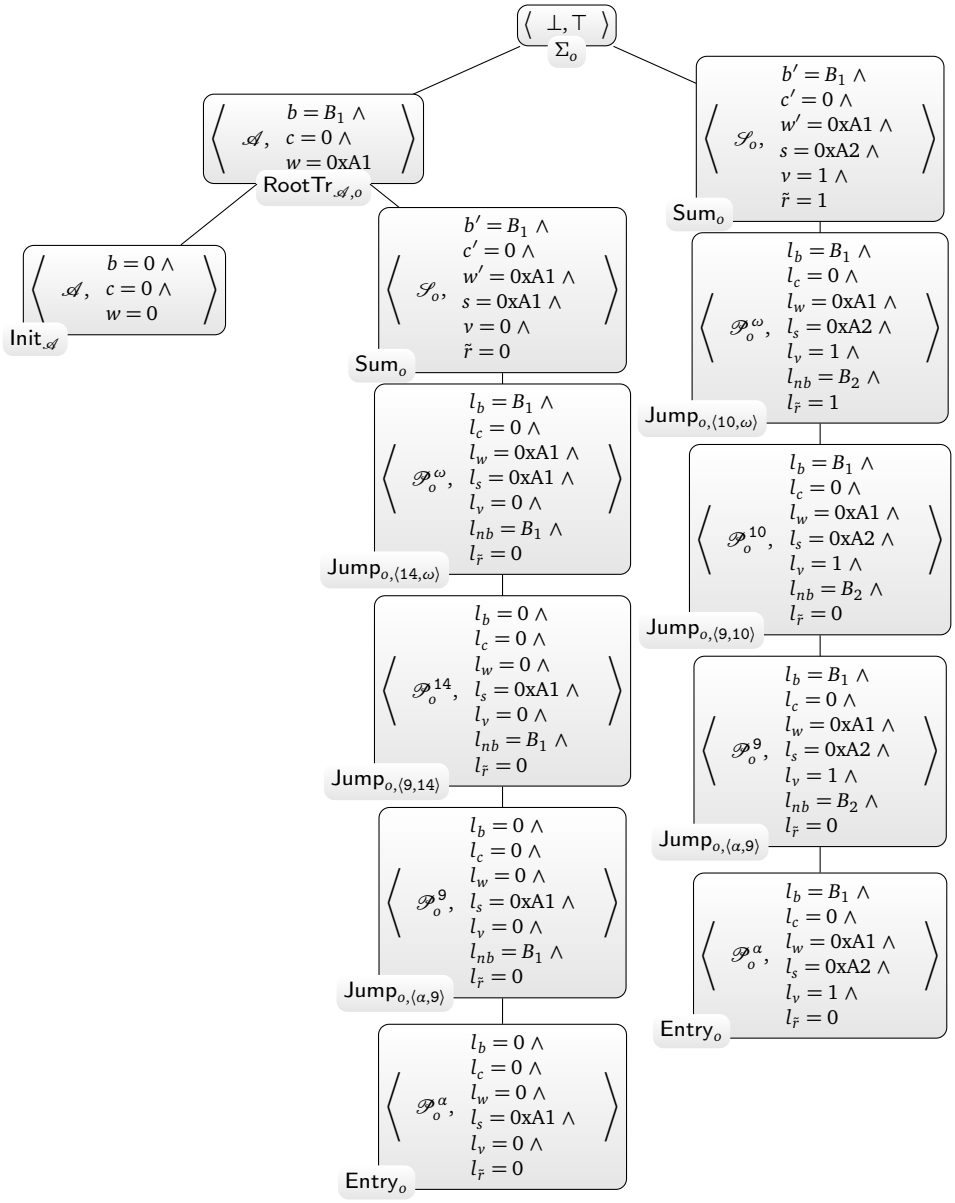


Figure 6.3. A refutation tree for the set of CHCs shown in fig. 6.2 modelling the contract `Auction` shown in fig. 6.1. Σ_o is the CHC $\mathscr{A} \wedge \mathscr{S}_o \wedge \tilde{r} \implies \bot$. $B_1 = 2^{256} - 5 \times 10^{15}$, and $B_2 = 2^{256} - 5 \times 10^{15} + 1$. The values 0xA1 and 0xA2 represent two ETHEREUM addresses.

ing CFG blocks of the function `offer` is $(b, c, w, s, v, l_b, l_c, l_w, l_s, l_v, l_{nb}, l_{\tilde{r}})$, where b, c and w respectively represent the state variables `bid`, `cash` and `winner`, s and v respectively represent the implicit function arguments `msg.sender` and `msg.value`, for each $x \in \{b, c, w, s, v\}$, l_x is the local copy of variable x, l_{nb} rep-

resents the local variable `new_bid`, and $l_{\tilde{r}}$ represents the local copy of the revert variable. The numbers in superscript for the predicates $\mathscr{P}_o^9, \mathscr{P}_o^{10}$, and $\mathscr{P}_o^{14}$ refer to the line numbers where the basic block that each predicate represents starts. The predicate $\mathscr{S}_o$ represents the summary of function `offer` and has the signature $(b, c, w, s, v, b', c', w', \tilde{r})$. The predicate $\mathscr{A}$ is the predicate representing the state of the contract `Auction` and has the signature (b, c, w). When a primed version of a variable appears in the body of a CHC, such variable is assumed to be applied primed in the head predicate.

The safety rule Σ_o for function `offer` is the CHC

$$\bot \leftarrow \mathscr{A} \wedge \mathscr{S}_o \wedge \tilde{r} \neq 0$$

The set of CHCs $\Pi_A \cup \{\Sigma_o\}$ is unsatisfiable, that is, there exist a refutation, shown in fig. 6.3, that proves Σ_o a contradiction given Π_A. In order to create the counterexample, the tree search provides the list of nodes $\langle \mathsf{Init}_{\mathscr{A}}, \mathsf{RootTr}_{\mathscr{A},o}, \Sigma_o \rangle$. The counterexample is constructed initially by using the node $\mathsf{Init}_{\mathscr{A}}$ to construct the contract by setting `bid=0`, `cash=0` and `winner=0`. Then, the following two nodes are used to create two transaction completing the counterexample, which functions and arguments are given in each node's children. In particular, the node $\mathsf{RootTr}_{\mathscr{A},o}$ is the result of a call to `offer()` having `msg.sender=0xA1` and `msg.value=0` represented by the child Sum_o, and the node Σ_o that fails an assertion is the result of a call to `offer()` having `msg.sender=0xA2` and `msg.value=1` represented by the child Sum_o. The counterexample shows that an initial null offer that results in a current bid of $2^{256}-5\times10^{15}$ wei (B_1 in fig. 6.3, 2.3×10^{61} USD at the time of writing), leaving `cash` null in line 15. The following transaction places a very small offer of 1 wei (2×10^{-16} USD at the time of writing), causing the the new bid (B_2 in fig. 6.3) to be higher than the previous. The attempt to refund the previous offer fails because of the assertion on line 10.

6.4 Implementation

A prototype of the proposed modelling approach has been implemented in collaboration with the engineers from the ETHEREUM Foundation, inside the SMTChecker component [smt20, AR18] of the Solidity compiler [Eth18b]. Specifically, the implementation of our work consists of the CHC model checking engine of SMTChecker, called SOLICITOUS.

The SOLICITOUS functionality can be enabled in the compilation by providing the corresponding *pragma* directive in the source file. Once enabled, the com-

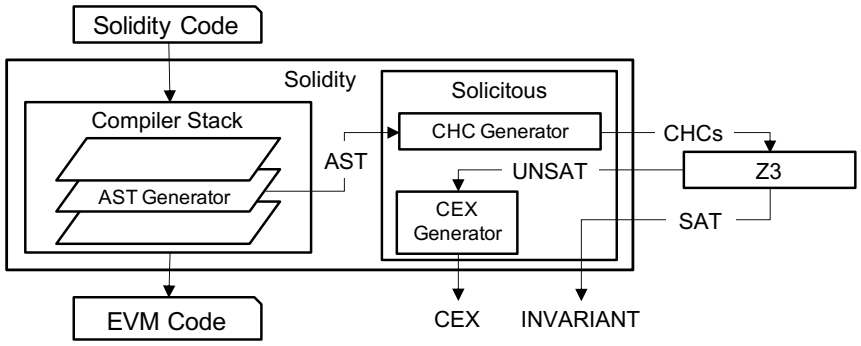


Figure 6.4. SOLICITOUS module inside the Solidity compiler.

piler provides the main Abstract Syntax Tree (AST) to SOLICITOUS that generates the CHC model of the contract following Algorithm 2. The CHC model is then provided to the engine SPACER [KGC16] of the SMT solver Z3 [dMB08] for solving. In case an assertion failure is detected, SOLICITOUS can provide a transaction trace as a witness to the failure, which can easily be checked by the developer. An overview of SOLICITOUS and Solidity can be seen in fig. 6.4.

The emphasis of this technique is in the modelling of the control flow of Solidity contracts. The control flow corresponds to AST nodes related to language constructs such as loops and conditional branches. Visiting these nodes triggers the creation of the corresponding clauses as described in section 6.2. In addition, the AST nodes corresponding to Solidity expressions result in accumulating the constraint ϕ of the clauses. Each expression node introduces a new SMT variable of the type of the expression. As an implementation detail, the unique identifiers the compiler assigns to AST nodes are used for guaranteeing unique names for these variables.

Solidity offers two special types of functions: *modifiers* and *constructors*. Modifiers represent pieces of code that envelope a function body. Therefore, modifiers' definitions depend on the functions they envelope, and they are not encoded separately but instead in-lined to the functions. Constructors define the initialization procedure executed at deployment time of a contract. The constructor modeling is prepended by providing the initialization $I(\boldsymbol{s})$ where variables are either zeroed or given their explicit initial values. In contracts that inherit base classes, the inheritance order is obtained by the Solidity compiler using the C3 linearization [BCH$^+$]. In addition, each constructor is executed exactly once. In our implementation, the entire deployment procedure, which might include the inheritance linearization and state variable initialization, is in-lined into a single constructor function.

SOLICITOUS currently supports a working subset of the Solidity language, in-

cluding the complex control flow and arithmetic operators (except exponentiation), integers of all available sizes, Boolean variables, arrays, mappings access and assignment, and inheritance. Strings and structs are currently not supported, and their occurrences in ϕ are replaced by nondeterministic operations in order to maintain soundness. Continuous support and the addition of the remaining language features is a goal of the ETHEREUM Foundation, and the supported subset of language is therefore expected to grow.

6.5 Experiments

The precision and language coverage of SOLICITOUS is evaluated on a set of real-world contracts over a 17 months period, between the block 7 million, mined 2nd of January 2019 and the block 10 million, mined on 4th of May 2020. All contracts deployed in that period that are written in Solidity v0.5 and v0.6, and are available through the Etherscan block explorer [eth20a] were extracted. The benchmarks are available at `https://scm.ti-edu.ch/repogit/verify-solidity-contracts.git`.

The query consisted of 1147850 addresses, obtaining 136802 contract sources, of which 27887 are unique: 367 v0.6, 10301 v0.5, and 17219 of previous versions. The tools was run only on contracts containing assertions. However, commented assertions were also checked. Commented assertions are of special interest because developers might have removed them before deployment in order to reduce gas cost, believing them to always hold. In total, the set consists of 6061 v0.5 contracts including 11076 assertions (the V5 benchmark set), and 77 v0.6 contracts including 163 assertions (the V6 benchmark set).

SOLICITOUS[1] is here compared against three other tools: SOLC-VERIFY [HJ19] and VERISOL [LCWD18] that verify Solidity source code, and MYTHRIL [Con18] that verifies Ethereum Virtual Machine (EVM) bytecode. MYTHRIL differs from the other tools in that it is a purely bounded checking engine of three transactions. Unlike SOLICITOUS, SOLC-VERIFY and VERISOL, MYTHRIL does not produce safe inductive invariants, and contracts MYTHRIL reports safe can be considered safe only up to three transactions after contract deployment. In this sense MYTHRIL can report only unsafe results, and only if a counterexample within three transactions exists. It is also hard to make claims about the validity of its counterexamples, as MYTHRIL authors do not provide any scientific publication that explains their technique. Despite its limitations, MYTHRIL is well known

[1] Available at `https://github.com/usi-verification-and-security/solc`

in the smart contracts community for having the best support for language features. In our comparative analysis, MYTHRIL serves as a gold standard for the language support metric. To the best of our knowledge these tools are the only ones with which an automated comparison is possible.[2] Both SOLC-VERIFY and VERISOL support only Solidity v0.5, thus for the comparative analysis using V5, it is used a legacy version of SOLICITOUS supporting v0.5 that has no support for counterexample generation. SOLC-VERIFY, VERISOL and legacy SOLICITOUS are sound but over-approximative. Specifically, while safe results are justified in these tools by an inductive invariant that proves safety, the tools do not justify unsafe results: in particular they do not provide an execution that would serve as a counterexample for the validity of an assertion. Therefore there is a distinction between 'not safe' and 'unsafe', using the former when no or spurious counterexample is produced and the latter when a concrete counterexample proves a real bug. The current SOLICITOUS implementation is separately evaluated using V6 to assess the concrete counter-example generation for proving unsafe results.

6.5.1 Counterexample Generation

The overall results for the V6 benchmark set is shown in table 6.1. SOLICITOUS is executed with two different types of encodings where integer arithmetic is encoded both without and with modularity. The former allows arbitrarily large values, while the latter models overflow and underflow precisely. MYTHRIL reports 13 safe contracts up to three transactions. SOLICITOUS performs the best over this benchmark set, not only guaranteeing a good number of contracts to be safe, but also supporting the language features present in most contracts. The counterexamples of the 7 unsafe contracts reported by SOLICITOUS were all checked to be concrete with the ETHEREUM evaluator HEVM [Eth20b]. Every counterexample leads to a runtime exception. Despite the small number of benchmarks due to Solidity v0.6 being very recent at the time of writing, our results show that SOLICITOUS is capable of generating valuable witnesses of assertion failures that can help developers to prevent vulnerabilities.

In addition to its standard execution, in which a potential assertion failure is reported by mentioning its line number in the source file, SOLICITOUS is also capable of generating concrete counterexamples to prove that the result is unsafe and not spuriously reported not safe due to the over-approximations of unsupported features. Unlike the fixed-size bounded approach of VERISOL and

[2] Two other tools were considered for the comparison, namely ZEUS [KGDS18] and SAFEVM [ACG+19], but ZEUS is not publicly available and SAFEVM only supports Solidity v0.4.

Table 6.1. Experimental results for the V6 benchmark set. INT and MOD stand for integer and modulo arithimetics. SOL and M respectively stand for SOLICITOUS and MYTHRIL. Verified shows the percentage of contracts with either Safe or Unsafe result.

	INT	MOD	
	SOL	SOL	M
Safe	32	27	–
Unsafe	7	7	1
Timeout	5	9	63
Error	33	34	0
Verified	50%	44%	18%

MYTHRIL, SOLICITOUS generates counterexamples of arbitrary length, reporting assertion failures that can happen at any point in the lifecycle of a contract.

6.5.2 Comparative Analysis

To get a better understanding of SOLICITOUS performance on a larger benchmark set, the 0.5 version of SOLICITOUS, SOLC-VERIFY, and VERISOL are evaluated on V5. The results are shown in table 6.2. Safe contracts are those for which all the assertions in the code are proved safe by safe inductive invariants. Not safe contracts have at least one assertion that is not proven safe. The timeout of each individual verification run is 60 seconds. Verification tasks halted for various types of errors are counted in the error row.

SOLICITOUS reports the largest amount of safe inductive invariants for both arithmetic encodings. Regarding the not safe results, SOLICITOUS can indistinguishably produce spurious and concrete results depending on whether unsupported features are present or not, since they are modelled as non-deterministic operations in order to preserve soundness. Similarly, SOLC-VERIFY introduces over-approximations during its translation to Boogie that produce the same effect. VERISOL presents the same issue, however if no invariant is found it performs a further step creating a bounded model of length four. If the bounded check reports unsafe, VERISOL produces a concrete counterexample. In summary, VERISOL can prove an assertion unsafe only if it can fail within four transactions after contract deployment. The unsafe reports proved by a concrete counterexample are shown with an asterisk in table 6.2.

The table also provides a comparison against MYTHRIL. Considering that

Table 6.2. Experimental results for the V5 benchmark set. INT and MOD stand for integer and modulo arithimetics. SOL, SV, VS, and M respectively stand for SOLICITOUS, SOLC-VERIFY, VERISOL and MYTHRIL. The Verified row shows the percentage of contracts reported either Safe or Not safe. The best result in each category is highlighted. * These numbers refer to unsafe reports proved by a concrete counterexample.

	INT			MOD			
	SOL	SV	VS	SOL	SV	VS	M
Safe	**1720**	778	135	**1681**	54	117	–
Not safe	142	572	298 (46*)	93	515	198 (31*)	23*
Timeout	586	89	**37**	678	**56**	130	5426
Error	3613	4622	5591	3609	5436	5616	**33**
Verified	**30%**	22%	7%	**29%**	9%	5%	9%

MYTHRIL can only produce bounded proofs, the number of contracts reported safe (579) is not reported in table 6.2. Our experiments show that SOLICITOUS is the tool that guarantees the largest amount of contracts to be safe, and that it is also the one able to verify the largest amount of contracts in general. Regarding the coverage of language features, using the amount of errors as a proxy metric, MYTHRIL possesses the best support. SOLICITOUS is closer to it than SOLC-VERIFY or VERISOL. Given the positive results, aligned with the practical nature of the benchmarks set used, SOLICITOUS stands as a valuable tool for Solidity developers.

6.6 Related Work

There is much interest in formally verifying ETHEREUM smart contracts, and several tools rely on different techniques to verify either Solidity or Vyper source code, or EVM bytecode. OYENTE [LCO+16] is one of the pioneers in this field, and uses symbolic execution of EVM bytecode to find common vulnerabilities. MYTHRIL [Con18] is a security tool based on control-flow analysis and concolic execution of EVM, supporting analysis of assertions up to a fixed bound of transactions. MAIAN [NKS+18] is also bounded in the number of transactions and searches EVM bytecode for three specific types of vulnerabilities. SECURIFY [TDDC+18] encodes EVM bytecode into Datalog to analyze programs, targeting specific types of bugs encoded as data patterns. VERX [PDT+20] verifies temporal properties written using a specification language for a particular class of contracts referred

as *effectively external callback free*. It requires user intervention when the automatic inference of abstraction predicates fails. The tool is not publicly available. MANTICORE [MMH+19] has a symbolic execution engine for EVM that uses SMT to systematically explore the state space of the contract by repeatedly executing *symbolic transactions*. KEVM [HSR+18] is a formal specification of the EVM semantics written in the K-FRAMEWORK [RS10]. It provides an assisted theorem prover and a specification language for further analysis, including reachability. Similarly, KVYPER [Fra18b] and KSOLIDITY [Fra18a] are the Vyper and Solidity semantics expressed over the the K-FRAMEWORK. KLAB [ELHL20] provides a specification language tailored for smart contracts that compiles to general K properties and a framework for proof debugging and counterexample analysis based on KEVM. SAFEVM [ACG+19] verifies EVM code produced by Solidity 0.4 through an intermediate translation to C that can be checked with three different backend C-verifiers. ZEUS [KGDS18] translates Solidity into LLVM bitcode which is fed to the SEAHORN [GKKN15] model checker. A subset of Solidity not including loops is verified after a translation to F* [BDLF+16]. WHY3 [Why18] has also been used to verify translated Solidity programs. However, Why3 does not support many of the Solidity constructs and is no longer developed. Slither [FGG19] translates Solidity to its own intermediate SSA language and performs bounded checks for several vulnerability classes. More recently, the tools SOLC-VERIFY [HJ19] from SRI and VERISOL [LCWD18] from Microsoft verify Solidity contracts using the language Boogie as intermediate representation.

6.7 Conclusions and Future Work

This chapter presented a formal technique for modelling smart contracts using CHCs. The results reported in this chapter have been published in [MOA+20]. The models are designed to formally capture semantic features specific for smart contracts, enable fully-automated verification of safety properties, and are suitable for exploiting sequential and parallel generic theorem provers in the task of analysis and contract invariants generation. The modelling technique is implemented for the Solidity language and its effectiveness is evaluated through an extensive experimentation involving 6138 contracts specifying 11239 safety properties, showing concrete effectiveness in real-world settings.

The results obtained so far opened several directions for future research. The following are the three research challenges related to the contribution of this chapter. (1) The control-flow graph creation can be performed in different ways

by varying the block size and the operations that trigger the creation of new blocks. An interesting research direction involves comparing different CFG building schemes and their performances. (2) Increasing the support of Solidity language features might open new challenges also with respect of which achieves the best performance. (3) Encoding gas usage in the CHCs model would enable reasoning over gas-related properties in an unbounded way. However how to perform such modelling effectively is not trivial.

Chapter 7

Bounded Gas Analysis for Smart Contracts

Users interact with ETHEREUM by submitting transactions that miners execute for a fee charged on-the-fly based on the complexity of the execution. The exact fee, measured in gas, in general depends on the unknown state of the contract, and predicting even its worst case is in principle undecidable. Uncertainty in gas consumption may result in inefficiency, loss of money, and, in extreme cases, in funds being locked for an indeterminate duration.

This chapter presents two methods for determining the exact worst-case gas consumption of a bounded transaction execution using methods influenced by symbolic model checking. ETHEREUM provides a Turing-complete execution environment, and therefore computing the worst-case gas consumption is undecidable. The protocol imposes, however, a maximum gas consumption for a block, making the computation in principle decidable. The challenge of computing the exact worst-case gas consumption of a transaction is addressed relying on highly efficient methods adapted from symbolic bounded model checking and using SMT solvers. The central concept introduced in this chapter is the *gas consumption path* (GCP). A GCP is a symbolic execution path that models concrete execution paths all having the same gas cost. The proposed techniques exhaustively examine all GCPs of a smart contract function using symbolic methods. The paths are identified in the high-level language Solidity and projected to the low-level EVM bytecode currently used in ETHEREUM. This approach has several advantages: Due to the combination of high and low-level representations we are able to be precise on the execution paths while maintaining exactness of the gas consumption. The approach is independent of the low-level representation, where gas consumed by the instructions might depend on protocol version, dif-

ferent compilers might produce different code, and even the assembly language is subject to change.

The rest of this chapter is organized as follows. Section 7.1 introduces the necessary preliminaries for the concepts used in the chapter. Sections 7.2 and 7.3 present the two algorithms, respectively the GCP enumeration and the function-oriented GCP enumaration. Section 7.4 provides the concrete results for the execution of both algorithms over an example contract. Finally, sections 7.5 and 7.6 respectively provides an overview of the related work and conclusions for this chapter.

7.1 Preliminaries

Table 7.1. Some EVM instruction costs [Eth18a]. The second half of the table lists examples of instructions whose cost depends on the context in which they are executed and the arguments provided.

Instruction	Gas	Description
`JUMPDEST`	1	Indicates a valid jump destination
`POP`	2	Pop from the stack
`PUSH`n	3	Push an n-bit item to stack
`ADD/SUB`	3	Arithmetic Operation
`LT/GT/SLT/SGT/EQ`	3	Arithmetic comparisons
`MLOAD/MSTORE`	3	Memory operations
`MUL/DIV/MOD`	5	Arithmetic Operations
`JUMP`	8	Unconditional jump to a location at the top of the stack
`JUMPI`	10	Conditional jump to a location at the top of the stack
`SLOAD`	200	Load from storage
`CALL`	700	Call a contract transaction with zero-valued arguments
`CALLVAL`	9,000	Call a contract transaction with non-zero valued arguments
`SSTORE`	5,000	Store a zero, or non-zero when previous value is non-zero
`SSTORE`	20,000	Store a non-zero when previous value is zero
`SSTORE`	15,000	Added to refund counter when storing a zero and previous value is non-zero.

The complexity of an ETHEREUM transaction is measured in its *gas consumption*. Each EVM instruction has an associated gas consumption, a measure that relates the instruction to its storage or execution cost. See table 7.1 for examples of some costs. In addition to instruction-specific costs, certain instructions and declarations affect the size of the memory local to a function, called the *active memory* [Eth18a]. Let a and b be the sizes of the active memory in bytes, respectively, before and after executing an instruction. The possible change incurs a cost or a refund defined as

$$\Delta C_{mem}(a,b) = 3 \cdot (a-b) + \left\lfloor \frac{a^2}{512} \right\rfloor - \left\lfloor \frac{b^2}{512} \right\rfloor.$$

To execute a transaction through a miner, a user provides a price he or she is willing to pay for a unit of gas in a currency called Ether, and the total amount of Ether that the transaction may consume. Assuming no errors are encountered while running the transaction and the amount paid for the actual gas consumption is sufficient, the transaction is carried out successfully. If carrying out the transaction requires more gas than what is provided, the execution is terminated without a refund.

Due to the memory model of EVM, in some cases the cost of an instruction depends on arguments of the instruction or the state of the contract when executing the instruction. For example:

- The instruction `SSTORE` writes into contract storage. The operation is costly in particular if a non-zero value is written to a storage location that previously contained a zero value. The EVM execution model contains a *refund counter* which is used for rewarding the user for executing instructions that make EVM less expensive. This is reflected in the case where `SSTORE` instruction writes a zero value to a location that previously held a non-zero value, resulting in a refund.

- The instruction cost of the instruction pair `CALL` and `CALLVAL` depend on their arguments. The instructions are used to call a transaction in another contract. While technically two different instructions, they can be interpreted as a single instruction from the perspective of a higher-level language. In this case the cost of a transaction depends on whether the values of the arguments passed in the call are zero.

The cost of a complete transaction in EVM is in part defined by the flow of control dictated by the EVM state, arguments, and the function code. Due to argument and environment dependence of instruction costs, the control flow graph

is not sufficient for determining the transaction cost. The control flow graph is generalized here to a *gas consumption graph* by adding new edges and nodes based on the instruction argument and environment dependence in a natural way, and call paths in the gas consumption graph *gas consumption paths* (GCP). All executions of a function that follow the same gas consumption path consume therefore equal amount of gas. Our approach aims at identifying a GCP that maximizes the gas consumption over all GCPs. Instead of working directly on EVM bytecode, we base the analysis on the higher-level Solidity language, arguably the most popular language for writing smart contracts at the time. Therefore the concept of GCPs is generalized here to Solidity GCPs. These are not in general the same for instance due to low-level optimizations available for EVM. As a result we do not attempt to compute the gas consumption on the Solidity code, but instead compute exact EVM gas consumption using concrete executions that are guaranteed to cover all Solidity GCPs.

We assume that the Solidity GCPs cover also all EVM GCPs. We want to emphasise this methodological choice as a potential threat to the validity of the results, and will reflect it in the theorems on correctness in the next sections.

To identify potentially different GCPs we employ bounded-model-checking techniques [BCCZ99] together with SMT solvers [dMB08, HMAS16, BCD+11, CGSS13], by operating on the static single assignment (SSA) level of Solidity where loops have been unwound up to a given limit. The approach can be made complete by increasing the unwinding limit since the ETHEREUM protocol imposes a maximum gas consumption for a transaction.

7.2 Gas Consumption Path Enumeration

This section presents an algorithm for enumerating symbolically Solidity GCPs based on the unwound SSA representation of smart contracts. While the number of GCPs is in general exponential in the size of the unwound SSA representation, due to the symbolic representation the algorithm runs in polynomial space.

We first give the translation of a Solidity contract to an unwound SSA (USSA) form in fig. 7.1 for an example program adapted from [FDHS15]. For brevity, fig. 7.1 (a) uses a pseudo-code resembling the Solidity language instead of the actual Solidity language.[1] The contract consists of functions `f` and g, where g calls `f`. Function g writes to the storage variable z and uses the solidity function `msg.sender.transfer` here abstracted simply as `transfer`(z). Function `f` does

[1] For a compilable Solidity contract see fig. 7.2.

operations on its arguments inside a loop, stores the result into a local variable, and returns the result after the computation.

The search for GCPs is done on the USSA form, given in fig. 7.1 (b). The form consists of a sequence of *guarded assignments* having the form $c \to b = e(x)$ or $c \to b =^s e(x)$, where c is a conjunction of Boolean-valued expressions, and $e(x)$ is an operation over variables x. We distinguish between assignments where the left side of the equality is a variable in memory ($=$) and a storage location ($=^s$) since depending on the values these have different costs (see table 7.1). Similarly the costs of some instructions depend on their arguments. For this purpose we define the function ArgCond that maps an instruction to its cost condition. For instance, $\text{ArgCond}(a + b) = \emptyset$, and $\text{ArgCond}(\texttt{transfer}(x)) = \{x = 0\}$. The cost implied by ΔC_{mem} only depends on the control flow path and therefore requires no special treatment.

The pseudo-code of the enumeration-based algorithm is given in algorithm 3. The algorithm takes as input an entry point function $f(\vec{v})$ and constructs the USSA starting from f, in-lining recursively all functions called from f (line 1). The USSA is then traversed to construct a set of Boolean expressions C by adding each conjunct from each guard c of the USSA assignment in lines 4–9. Additional Boolean expressions are added to C for each storage assignment $=^s$ (line 7), and for each instruction whose cost depends on its arguments (line 9). The function $pre(x_i) = x_{i-1}$ maps a USSA variable x_i to its previous instantiation. In case x_i is the first instantiation (i.e., $i = 1$), $pre(x_i)$ is a "fresh" variable not appearing in the USSA.

In the second phase the algorithm exhaustively queries the SMT encoding of the USSA form for each Boolean combination of expressions from C and obtains values for $\vec{v}$ and $\mathbb{S}$ that cover these cases in case of satisfiability. The cost of each value combination for $\vec{v}$ and $\mathbb{S}$ is then queried by simulating the transaction, and the highest gas estimate is returned as the exact worst-case bound.

Running algorithm 3 on the USSA form on fig. 7.1 (b) gives

$$\begin{aligned} C = \quad & \{x_1 \geq y_1, y_1 \geq 0, (x_1 + y_1 = 0) \wedge (z_0 = 0), (x_1 + y_1 = 0) \wedge (z_0 \neq 0), z_1 = 0, \\ & \mathsf{f}_{i_1} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}, \mathsf{f}_{i_1} < \mathsf{f}_{a_1}, \mathsf{f}_{i_6} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}, \mathsf{f}_{i_6} < \mathsf{f}_{a_1}, \\ & (\mathsf{f}_{ret_1} = 0) \wedge z_1 = 0), (\mathsf{f}_{ret_1} = 0) \wedge z_1 \neq 0)\}, \end{aligned}$$

where the first two constraints $x_1 \geq y_1$ and $y_1 \geq 0$ and the whole of the second row constraining the local variables of the functions $f_{i_j}, f_{a_j}, f_{b_j}$ come from the if-conditions; the conjunctive constraints $(x_1 + y_1 = 0) \wedge (z_0 = 0), (x_1 + y_1 = 0) \wedge (z_0 \neq 0)$ come from the argument and environment dependency of SSTORE (see table 7.1), and the constraint $z_1 = 0$ comes from the argument dependency

```
1  contract C:
2    int z;
3    function g(x, y):
4      if (x >= y)
5        if (y >= 0)
6          z = x + y
7        transfer(z)
8      z = f(x, y)
9
10   function f(a, b):
11     int i = 0
12     while (i < a + b):
13       if (i < a):
14         i = i + a
15       else:
16         i = i + b
17     return i
```

(a) Pseudo-Solidity contract

$$x_1 \geq y_1 \wedge y_1 \geq 0 \rightarrow z_1 =^s x_1 + y_1; \tag{7.1}$$
$$x_1 \geq y_1 \rightarrow \texttt{transfer}(z_1); \tag{7.2}$$
$$true \rightarrow \mathsf{f}_{a_1} = x_2; \tag{7.3}$$
$$true \rightarrow \mathsf{f}_{b_1} = y_2; \tag{7.4}$$
$$true \rightarrow \mathsf{f}_{i_1} = 0; \tag{7.5}$$
$$(\mathsf{f}_{i_1} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \wedge (\mathsf{f}_{i_1} < \mathsf{f}_{a_1}) \rightarrow \mathsf{f}_{i_2} = \mathsf{f}_{i_1} + \mathsf{f}_{a_1}; \tag{7.6}$$
$$(\mathsf{f}_{i_1} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \wedge (\mathsf{f}_{i_1} \geq \mathsf{f}_{a_1}) \rightarrow \mathsf{f}_{i_3} = \mathsf{f}_{i_1} + \mathsf{f}_{b_1}; \tag{7.7}$$
$$(\mathsf{f}_{i_1} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \rightarrow \mathsf{f}_{i_4} = \texttt{ite}((\mathsf{f}_{i_1} < \mathsf{f}_{a_1}), \mathsf{f}_{i_2}, \mathsf{f}_{i_3}); \tag{7.8}$$
$$(\mathsf{f}_{i_1} \geq \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \rightarrow \mathsf{f}_{i_5} = \mathsf{f}_{i_1}; \tag{7.9}$$
$$true \rightarrow \mathsf{f}_{i_6} = \texttt{ite}((\mathsf{f}_{i_1} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}), \mathsf{f}_{i_4}, \mathsf{f}_{i_5}); \tag{7.10}$$
$$(\mathsf{f}_{i_6} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \wedge (\mathsf{f}_{i_6} < \mathsf{f}_{a_1}) \rightarrow \mathsf{f}_{i_7} = \mathsf{f}_{i_6} + \mathsf{f}_{a_1}; \tag{7.11}$$
$$(\mathsf{f}_{i_6} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \wedge (\mathsf{f}_{i_6} \geq \mathsf{f}_{a_1}) \rightarrow \mathsf{f}_{i_8} = \mathsf{f}_{i_6} + \mathsf{f}_{b_1}; \tag{7.12}$$
$$(\mathsf{f}_{i_6} < \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \rightarrow \mathsf{f}_{i_9} = \texttt{ite}((\mathsf{f}_{i_6} < \mathsf{f}_{a_1}), \mathsf{f}_{i_7}, \mathsf{f}_{i_8}); \tag{7.13}$$
$$(\mathsf{f}_{i_6} \geq \mathsf{f}_{a_1} + \mathsf{f}_{b_1}) \rightarrow \mathsf{f}_{i_{10}} = \mathsf{f}_{i_6}; \tag{7.14}$$
$$true \rightarrow \mathsf{f}_{i_{11}} = \texttt{ite}((\mathsf{f}_{i_6} \leq \mathsf{f}_{a_1} + \mathsf{f}_{b_1}), \mathsf{f}_{i_9}, \mathsf{f}_{i_{10}}); \tag{7.15}$$
$$true \rightarrow \mathsf{f}_{ret_1} = \mathsf{f}_{i_{11}}; \tag{7.16}$$
$$true \rightarrow z_2 =^s \mathsf{f}_{ret_1}; \tag{7.17}$$

(b) USSA approximation (bound = 2)

Figure 7.1. Converting a contract into a USSA

Input : Entry function f; unwind limit n
Output: A set of Boolean expressions C

Let $U =$ the USSA form starting from f unwound up to n
Let $C = \emptyset$
foreach *guarded assignment* $a \in U$ **do**
    Let $c_1 \wedge \ldots \wedge c_k$ be the guard of a
    $C = C \cup \bigcup_{i=1}^{k} \{c_i\}$
    if a *is of form* $c_1 \wedge \ldots \wedge c_k \rightarrow y =^s e(x)$ **then**
        $C = C \cup \{(e(x) = 0) \wedge (pre(y) = 0), (e(x) \neq 0) \wedge (pre(y) = 0)\}$
    end
    $C = C \cup \text{ArgCond}(e(x))$
end
foreach *truth value combination for the elements of* C **do**
    if $C \wedge U$ *is satisfiable* **then**
        Measure the gas consumption of f on environment corresponding to the satisfying truth assignment
        Update the maximum if necessary
    end
end

Algorithm 3: Enumeration-based algorithm to compute GCPs of a function f.

of CALL and CALLVAL, that is, ArgCond(transfer(z_1)); and the third row comes similarly from the argument and environment dependency of SSTORE.

The constraint set C is then provided to an SMT solver together with an SMT representation of the USSA form. Each combination of truth values for the constraints in C is queried from the USSA form, resulting in the worst case $2^{11} = 2048$ SMT queries. Note that due to the incremental implementation of SMT solvers in practice the number of queries might be (exponentially) smaller, depending on the order of the queries. In certain scenarios also the input $\vec{v}$ of the function might be known, reducing the number of queries to a fraction of the worst case.

From the results of the satisfiable queries the algorithm will extract concrete values for $\vec{v}$ and $\mathbb{S}$, which are then used for computing exact gas consumptions for the corresponding gas consumption paths.

The USSA form presented in fig. 7.1 does not acknowledge the invariant $z \geq 0$, and is therefore more permissive than the original contract. Obtaining such *contract invariants* is non-trivial and out of the scope of this chapter. To obtain exact worst-case gas consumption, contract invariants need to be conjoined to the USSA.

By construction of algorithm 3 and the definition of GCPs, we immediately have the following theorem:

Theorem 2 *Given a function f, assuming a USSA for f that exactly describes the contract behaviour, and that there is a one-to-one mapping between the Solidity and the EVM code, algorithm 3 return the worst-case gas consumption of f.*

7.3 Function-Oriented GCP Enumeration

This section presents an algorithm for *Function-Oriented GCP Enumeration* (FGCP), an approach to computing GCPs that prunes locally the immediately unsatisfiable gas consumption paths. The basic GCP Enumeration presented in section 7.2 in-lines every function call and computes GCPs from the encoding of the whole program. The function-oriented approach computes the paths gradually, starting from the low-level instructions and refining the set of GCPs discovered so far in a recursive manner. We expect local pruning of GCPs to be particularly efficient for contracts that call a given function multiple times, since the approach is able to reuse previously computed, function-specific GCPs.

To present the function-oriented approach, we change slightly the notation used in section 7.2. We introduce *cost equivalence classes* that extend the notion

of cost condition from a single instruction to a block of instructions and user-defined functions. The cost equivalence classes capture the conditions under with a function behaves differently with respect to gas consumption. They correspond exactly to the GCPs of the function. We use the term function to refer to both low level instructions, such as arithmetic operations, and user-defined functions, since cost-equivalence classes do not distinguish between the two. We do not distinguish between $=$ and $=^s$, but instead introduce a separate function SSTORE that is used for updating the storage $\mathbb{S}$. Finally, we introduce a separate function-oriented version of the static single assignment form, called FSSA, that is based on guarded function calls instead of guarded assignments.

Definition 15 (Environment) *Given a function $f(\vec{v})$ and storage $\mathbb{S}$, the environment of an execution of f is an evaluation υ for $\vec{v}$ and σ for $\mathbb{S}$.*

Given a function f and its environment, the execution of f is deterministic and results in a new storage state.

Definition 16 (Cost-equivalence class) *Given a function $f(\vec{v})$ and storage $\mathbb{S}$, a cost-equivalence class is a formula representing environments $\varphi(\mathbb{S}, \vec{v})$, such that the cost of executing f on any environment satisfying φ is the same.*

Algorithm 4 computes a set of cost-equivalence classes for the input function $f(\vec{v})$. Note that the set of classes computed by the algorithm is not guaranteed to be the minimal, namely there may be different classes representing executions with equal costs.

We define with $\mathbb{C}$ the map from function to a set of its cost-equivalence classes, such that every environment satisfies exactly one formula. Thus, given a function $f(\vec{v})$, the cost equivalence classes of f is the finite set

$$\mathbb{C}[f(\vec{v})] = \{\varphi_1(\mathbb{S}, \vec{v}), \ldots, \varphi_n(\mathbb{S}, \vec{v})\}$$

such that $\bigvee_{i=1}^{n} \varphi_i$ is a tautology and for all $i \neq j, \varphi_i \wedge \varphi_j$ is unsatisfiable.

Initially, all the basic functions are defined in $\mathbb{C}$ having their classes inserted manually following their cost specification. For instance, in ETHEREUM storing a value in the storage is performed by the operation SSTORE, which cost depends on both the value and the storage location [Eth18a]. In particular, setting a storage location from zero to a non-zero value costs more than all the other cases. Thus, according to the EVM gas consumption specifications, $\mathbb{C}[\text{SSTORE}(l, v)] = \{(\mathbb{S}[l] = 0 \wedge v \neq 0), (\mathbb{S}[l] \neq 0 \vee v = 0)\}$.

Input : A FSSA $f(\vec{v})$, the cost-equivalence classes $\mathbb{C}$.
1 Let $Tr_f(\mathbb{S}, \vec{v})$ the USSA of f, having local SSA variables $\vec{l}$.
2 **foreach** $c \rightarrow g(\vec{l} \mapsto \vec{v_g})$ *in* f **do**
3 **with** Tr_f **compute**
4 $\pi(\mathbb{S}, \vec{v}) :=$ path constraint of the call $g(\vec{v_g})$.
5 $M(\mathbb{S}, \vec{v}, \vec{v_g}) :=$ the mapping from $\vec{v}$ to $\vec{v_g}$ of the call $g(\vec{v_g})$.
6 **end**
7 Let $s = \emptyset$
8 **foreach** $\varphi(\mathbb{S}, \vec{v})$ *in* $\mathbb{C}[f]$ **do**
9 **if** $\neg\pi \wedge \varphi$ *is* SAT **then** $s \leftarrow s \cup \{\neg\pi \wedge \varphi\}$;
10 **foreach** $\psi(\mathbb{S}, \vec{v_g})$ *in* $\mathbb{C}[g]$ **do**
11 Let $\varphi'(\mathbb{S}, \vec{v}) = \pi \wedge \varphi \wedge M \wedge \psi$
12 **if** φ' *is* SAT **then**
13 $s \leftarrow s \cup \{\varphi'\}$
14 **end**
15 **end**
16 **end**
17 $\mathbb{C}[f] \leftarrow s$
18 **end**

Algorithm 4: The FGCP algorithm to compute the set $\mathbb{C}[f]$ of cost equivalence classes of f.

Algorithm 4 assumes that $\mathbb{C}$ contains all the functions in the input function's call tree. Such functions are both basic functions and user defined functions for each of which a previous execution of the algorithm created its classes. We assume there is no recursion.

Definition 17 (FSSA: Function-oriented SSA) *Given a function $f(\vec{v})$ and its USSA representation, the FSSA representation is a list of guarded function calls, one for each function call in f and having the form $c \rightarrow g(\vec{l} \mapsto \vec{v_g})$ where $\vec{l} \supseteq \vec{v}$ are the local USSA variables representing the inlining of the call mapped to the arguments $\vec{v_g}$ needed for executing g, and $c \in \vec{l}$ is the USSA guard of the call.*

The FSSA provides the necessary information for building the *call specific* mapping M on line 5 of algorithm 4. In particular, M maps the current call site to the previously computed cost-equivalence classes of the callee. Therefore M enables building the cost-equivalence classes of a callee function g (from definition 16 defined over its variables $\vec{v_g}$), in terms of $\vec{v}$. A new cost-equivalence class in terms of the caller variables is built by conjoining M and ψ in line 11, resulting in a formula defined over $\mathbb{S}$ and $\vec{v}$. Such operation is always possible because the USSA provides a formula for computing USSA local variables $\vec{l}$ in terms of $\vec{v}$. Then, a simple rewriting following each FSSA call $\vec{l} \mapsto \vec{v_g}$ will therefore build the new class in terms of $\vec{v}$. An example of FSSA is given in fig. 7.2.

Theorem 3 *Given a function f, assuming that the USSA formula Tr_f used in algorithm 4 exactly describes the contract behaviours and that the EVM and Solidity gas consumption paths have one-to-one correspondence, algorithm 5 returns the maximum gas consumption of f.*

Theorem 3 ensures that the size of each classes set in $\mathbb{C}$ is finite, and that every possible behaviour is considered. This proves termination and completeness of the algorithm.

Proof sketch. The property that every environment satisfies exactly one class in $\mathbb{C}$ is an invariant during the execution of algorithm 4. The property is maintained inductively. In line 11 the algorithm creates the new classes φ' for f from the classes ψ of the callee g. Each φ' is mutually exclusive provided that all ψ in $\mathbb{C}[g]$ are mutually exclusive, because every ψ appears in the conjunction. Furthermore, the disjunction of s is a tautology meaning it is complete, if the disjunction of all ψ in $\mathbb{C}[g]$ is also complete. The models excluded by π being in the conjunction in line 11, are considered by the class $\neg\pi$ added to s in line 9. □

Input : A function $f(\vec{v})$, the cost-equivalence classes $\mathbb{C}$.
Output : The maximum cost c.
1 Let $c = 0$
2 **foreach** $\varphi(\mathbb{S}, \vec{v})$ *in* $\mathbb{C}[f\,]$ **do**
3 | Let $\sigma(\mathbb{S}), \upsilon(\vec{v}) =$ an environment in φ
4 | Let c' the cost of executing $f(\upsilon)$ with storage σ
5 | **if** $c' > c$ **then** $c \leftarrow c'$;
6 **end**
7 **return** c

Algorithm 5: The algorithm to compute the maximum gas consumption.

Algorithm 5 computes the costs of every cost-equivalence class and returns the maximum. Definition 16 ensures that every environment satisfied by the same equivalence class has the same cost. Thus, on line 3 the SMT solver is queried for a model of each class φ, which is guaranteed to be satisfiable by line 13 of algorithm 4. We split the environment in two parts: σ assigning storage locations' values, and υ assigning values to the input argument $\vec{v}$. Then on line 4 the function f is executed on the specific environment and the cost of such execution is returned. If the cost is higher than the current maximum, on line 5 the current maximum is updated to the new value.

Parallelization Opportunities

Often the complexity and intrinsic sequentiality of model checking algorithms prevent parallelization. This results in missing the opportunity to exploit the modern hardware infrastructures, increasingly directed toward higher degrees of parallelism. Algorithms 3, 4 and 5 are immediately suitable for parallelization.

Due to the worst-case exponential number of SMT queries that Alg. 3 needs to perform we believe that the part most profiting from parallelization is the evaluation of truth assignments and simulating the execution on the block starting at line 11. Since the USSA form U remains the same over the queries, the parallelization may be enhanced with a clause-sharing scheme similar to [MHS16].

Algorithm 4 can be parallelized by asynchronously executing the building of all formulas and SMT queries inside the **foreach** at line 8. Each independent process can safely execute line 13 because inserting a new formula in the set s affects neither the future nor running executions. Executing line 16 and proceeding to the next function call can be done as soon as all independent executions are terminated. Algorithm 5 can be easily parallelized with using the MapReduce paradigm by defining proper *map* and *reduce* procedures. In this particular case the procedure *map* maps classes to their costs, while reduce compares the costs

```
1  contract C {
2    int a;
3    function f(bool c, int z)
4    {
5      if (c)
6      {
7        g(z);
8        z = z + 1;
9        g(z);
10     }
11   }
12
13   function g(int u)
14   {
15     a = u;
16   }
17 }
```

```
1  f(bool c, int z):
2    c → g(z ↦ u)
3    c → ADD(z ↦ x, 1 ↦ y, z1 ↦ r)
4    c → g(z1 ↦ u)
5
6  g(int u):
7    ⊤ → SSTORE(id(a) ↦ l, u ↦ v)
8
9  ADD(int x, int y, int r):
10   ⊤ → r = x + y
11
12 SSTORE(int l, int v):
13   ⊤ → 𝕊[l] = v
```

Figure 7.2. Left: An example contract with two functions. Right: the encoding to FSSA. Lines not representing an implication are only intended to show which function the following implications refer to. The macro `id()` returns the storage id of the variable.

in order to compute the maximum.

7.4 Example

In this section we provide the example contract `C`, and we simulate the execution of algoritms 3 and 4 on the contract `C` from fig. 7.2.

7.4.1 Function-Oriented GCP Enumeration

The contract in fig. 7.2 uses two basic functions, namely `ADD` and `STORE`. Following the ETHEREUM gas specification we define

$$\mathbb{C}[\texttt{ADD}(x,y,r)] = \{\top\}, \text{and}$$
$$\mathbb{C}[\texttt{SSTORE}(l,v)] = \{(\mathbb{S}[l] = 0 \wedge v \neq 0), (\mathbb{S}[l] \neq 0 \vee v = 0)\}.$$

The execution of algorithm 4 on $\mathtt{g}(u)$ and $\mathbb{C}$ will result in the classes

$$\mathbb{C}[\mathtt{g}(u)] = \{M_7 \wedge (\mathbb{S}[l] = 0 \wedge v \neq 0), M_7 \wedge (\mathbb{S}[l] \neq 0 \vee v = 0)\}.$$

where $M_7(\mathbb{S}, u, l, v) \coloneqq (l = \mathtt{id}(a) \wedge v = u)$. Note that M_7 describes the mapping of the specific function call in line 7 of the FSSA in fig. 7.2, which is the only function call in $\mathtt{g}$, having path constraint $\pi \coloneqq \top$. The transition relation $Tr_{\mathtt{g}}$ of $\mathtt{g}$ is

$$Tr_{\mathtt{g}}(\mathbb{S}, u) \coloneqq \mathbb{S}[\mathtt{id}(a)] = u.$$

After simplifying, the classes of $\mathtt{g}$ are

$$\mathbb{C}[\mathtt{g}(u)] = \{(\mathbb{S}[\mathtt{id}(a)] = 0 \wedge u \neq 0), (\mathbb{S}[\mathtt{id}(a)] \neq 0 \vee u = 0)\}.$$

We now consider an execution of algorithm 4 on $\mathtt{f}(c, z)$, a function with 3 FSSA guarded calls at lines 2, 3 and 4 of fig. 7.2 right, all having $\pi \coloneqq c$. The USSA transition relation of $\mathtt{f}$ is

$$Tr_{\mathtt{f}}(\mathbb{S}, c, z) \coloneqq c \rightarrow (\mathbb{S}[\mathtt{id}(a)]_1 = z \wedge z_1 = z + 1 \wedge \mathbb{S}[\mathtt{id}(a)]_2 = z_1),$$

and the mappings for each function call in $\mathtt{f}$ are

$$\begin{aligned} M_2(\mathbb{S}, c, z, u) &\coloneqq (u = z), \\ M_3(\mathbb{S}, c, z, x, y, r) &\coloneqq (x = z \wedge y = 1 \wedge r = z + 1), \text{and} \\ M_4(\mathbb{S}, c, z, u) &\coloneqq (u = z + 1). \end{aligned}$$

The resulting classes for $\mathtt{f}$ are

$$\begin{aligned} \mathbb{C}[\mathtt{f}(c, z)] = \{ \quad & \neg c, \\ & c \wedge \mathbb{S}[\mathtt{id}(a)] = 0 \wedge z \neq 0 \wedge z \neq -1, \\ & c \wedge \mathbb{S}[\mathtt{id}(a)] = 0 \wedge z = 0, \\ & c \wedge \mathbb{S}[\mathtt{id}(a)] = 0 \wedge z = -1, \\ & c \wedge \mathbb{S}[\mathtt{id}(a)] \neq 0\}. \end{aligned}$$

Algorithm 4 computes a total of 5 classes. This set is not the minimal because both classes $\mathbb{C}[\mathtt{f}]_3$ and $\mathbb{C}[\mathtt{f}]_4$ cause exactly one write from zero to non-zero, resulting in the same cost. The minimal set would then be of size 4. However, by trivially combining all the cases, the total number of combinations is 16. The proposed algorithm is therefore able to reduce the number of possible classes consistently with respect to trivial enumeration, keeping the size of $\mathbb{C}$ reasonable.

7.4.2 Symbolical GCP enumeration

The USSA form for contract C in fig. 7.2 is

$$
\begin{aligned}
&c_1 \to \mathtt{g}_{u_1} = z_1;\\
&c_1 \to a_1 =^s \mathtt{g}_{u_1};\\
&c_1 \to \mathtt{g}_{u_2} = z_1 + 1;\\
&c_1 \to a_2 =^s \mathtt{g}_{u_2};\\
&a_3 = \mathtt{ite}(c_1, a_2, a_0);
\end{aligned}
$$

Running algorithm 3 on the USSA gives the set

$$
\begin{aligned}
C = \quad &\{c_1, (a_0 = 0) \wedge (\mathtt{g}_{u_1} = 0), (a_0 \neq 0) \wedge (\mathtt{g}_{u_1} = 0),\\
&(a_1 = 0) \wedge (\mathtt{g}_{u_2} = 0), (a_1 \neq 0) \wedge (\mathtt{g}_{u_2} = 0)\}.
\end{aligned}
$$

The size of the set is five, resulting in the worst case $2^5 = 32$ SMT queries.

7.5 Related work

The tool GASPER [CLLZ17] analyses ETHEREUM smart contracts compiled into the low-level EVM bytecode and is capable of identifying certain constructs that are costly and can be simplified to equivalent, less costly programs. Similarly, the tool GASOL [ACG+20] provides an ECLIPLE plugin for Solidity that provides the user with an upper bound for the cost of a target function. Furthermore, GASOL is able to detect certain under-optimized storage patterns, automatically proposing optimal solutions.

Incorrect gas consumption values for EVM instructions enable DoS attacks on ETHEREUM based on frequently executing under-evaluated instructions. In [CLW+17], the authors propose an emulation-based framework to automatically adjust the gas prices of EVM instructions based on measuring their resource consumptions. As part of the emulation the approach measures the gas consumption of functions based on control and data flow, but the emulation is based on random sampling and therefore is bound to be incomplete for all but the simplest contracts.

In [GKJ+18] the authors propose a static analysis tool called MADMAX to automatically detect gas-related vulnerabilities in ETHEREUM that trigger undesired behaviors when a transaction runs out of gas. The authors evaluated MADMAX over smart contracts deployed in ETHEREUM up to April 2018 and it reported vulnerabilities in contracts that hold together approximately 2.8 Billion USD. A

manual inspection of a sample resulted in 81% of the reported contracts to be actually vulnerable.

Correctness aspects of smart contracts other than gas consumption have been studied using symbolic methods. For instance Oyente [LCO+16] extracts the control flow graph from the EVM bytecode of a contract, and symbolically executes it in order to detect some vulnerability patterns, although it is neither sound nor complete. Zeus [KGDS18] is a framework for verification of Solidity smart contracts using abstract interpretation and symbolic model checking. The tool works by converting Solidity to LLVM bit code, and verifying reachability properties using the SeaHorn model checker [GKKN15].

7.6 Conclusions and Future Work

This chapter presents the two algorithms for addressing the problem of estimating the gas consumption of ETHEREUM smart contracts, based on techniques inspired by bounded model-checking. The contributions presented in this chapter has been published in [MBH+18]. The first, Gas Consumption Path Enumeration, collects all constraints that affect the gas consumption, evaluates all combinations of them one-by-one, and simulates those that are satisfiable. The second, Function-Oriented Gas Consumption Path Enumeration constructs GCPs for each function as explicit *cost-equivalence classes*, which are reused through variable renaming to recursively construct more cost-equivalence classes for calling functions. Both algorithms can be parallelized, and this chapter reports possible ways to improve performance in that way. In summary, the contributions of this chapter are the definition of GCP and an algorithm to enumerate all paths from a bounded symbolic representation of a smart contract; the definition of cost-equivalence class for smart contracts functions and an algorithm for enumeratation; and a comparative analysis with an example. The most interesting future research steps are the implementation and evaluation of the techniques on real-world smart contracts, and the integration with existing model checking technology for smart contracts.

Chapter 8

Conclusions

Formal methods address the problem of proving system correctness by producing a logical argument stating that any possible behaviours manifestable by the system are correct and do not produce errors. Automated formal verification aims to perform such task without human intervention. Excluding human errors from altering the verification result comes with the downside of increasing the complexity of the underlying algorithms, that are entirely entrusted to perform the reasoning. Therefore, full automation of verification introduces important challenges in devising efficient reasoning methods.

This thesis addressed the problem of automated verification efficiency by proposing methods to improve the automatic reasoning capabilities of verification techniques for software systems. The proposed contributions target both the two automatic tasks performed by model checkers in the process of automatic verification: modelling and checking.

Chapters 3 and 4 presents methods to exploit the massive computational power offered by distributed computing systems in the task of checking a mathematical model. The parallel execution is performed using multi-agent solvers that cooperate by exchanging data, in order to actively and collectively converge to a solution while preventing the same tasks to be performed redundantly in different solvers. In particular, chapter 3 focuses on techniques to parallelize SMT solving that have a direct impact of bounded model checking and optimization. The results from this track have been published in [MHS16, HMS15]. Chapter 4 centre on the parallelization of the inductive-based reasoning IC3 [Bra11] using SMT, completing the picture by providing parallel methods suitable also for unbounded model checking. This contribution has been published in [MGHS17]. The efforts invested in the design of parallel methods for bounded and unbounded model checking performed over big distributed com-

puting environment resulted in the SMTS framework, presented in chapter 5 and published in [MHS18]. SMTS is a tool that provides effective primitives to facilitate the parallelization of sequential solvers. This ability is demonstrated in chapters 3 and 4, and in the related publication [BHMS20] where the unbounded model checking algorithm PD-KIND [JD16] is parallelized using SMTS.

Chapters 6 and 7 focus on the design of modelling techniques necessary to correctly and effectively verify the new emerging technology of smart contracts running on blockchain systems. A proper modelling is crucial for achieving optimal performance later during the checking task. In particular, chapter 6 focuses on the modelling of ETHEREUM smart contracts written in Solidity using CHCs. This contribution provides an effective solution for checking safety of real-world smart contracts using existing inductive-based reasoning, including those parallelized in chapter 4. This allows smart contracts verification to directly benefit from such research effort and exploit the massive computational power offered by distributed computing clusters to improve efficiency. The results for this contribution are published in [MOA+20]. Finally, chapter 7 studies the problem of finding the worst-case gas consumption of smart contracts transactions, presenting two algorithmic solutions specific for ETHEREUM. This contribution is published in [MBH+18].

Future work directions from this thesis centre around further improvements of model checking performance. The most important direction for parallelization is making cooperative multi-agent techniques scalable up to thousands of solvers. This task poses several challenges for its accomplishment that require dealing with many peculiarities that, despite being well observed in this thesis, are still not clear yet. Such phenomena regard diversification and cooperation. In particular, experiments from chapter 3 show that fixing atoms makes the partitions sometimes harder for T-DPLL solvers and sharing clauses between partitions is not beneficial for the performance. Similarly, experiments from chapter 4 suggest that partitioning is not always beneficial for inductive-based solvers, and despite the number of exchanged lemmas is lower than the SMT systems, scaling to thousands of agent would result in huge burden. Clearing these points is highly non-trivial and would help engineering more effective parallelization techniques. The results so far are very promising, showing that further improvement could be substantial for parallel model checking and that SMTS can be used to pursue these research tracks thanks to its flexibility.

Smart contracts and blockchain systems are still a new and emergent area of informatics. Safety properties check and gas consumption analysis for smart contracts, presented respectively in chapters 6 and 7, are very interesting problems where formal methods are currently needed. It is likely that in the near future

new areas will emerge, requiring formal methods for more and more tasks in this context. However, many improvements can already be pursued. The most important is improving the support of language features for the CHCs encoding. This might reveal the need of further SMT theories in the bodies of the CHCs, requiring solvers to support them and potentially opening other challenges. Several small tweaks on the modelling, e.g. control-flow block size, can have huge impact on the solving task. Further knowledge on what is behind each modelling choice can improve performance dramatically. Finally, unbounded gas analysis would provide developers with more detailed information about the general efficiency of the functions, and how parameters affect the gas consumptions.

Bibliography

[ACG+19] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: a safety verifier for ethereum smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019*, pages 386–389, 2019.

[ACG+20] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. GASOL: gas analysis and optimization for ethereum smart contracts. In *Tools and Algorithms for the Construction and Analysis of Systems - 26th International Conference, TACAS 2020, Proceedings*, volume 12079 of *LNCS*, pages 118–125. Springer, 2020.

[AHJP14] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette. Dolius: A distributed parallel SAT solving framework. In *POS-14. Fifth Pragmatics of SAT workshop, a workshop of the SAT 2014 conference, part of FLoC 2014 during the Vienna Summer of Logic*, volume 27 of *EPiC Series*, pages 1–11. EasyChair, 2014.

[AR18] Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Proceedings*, pages 376–388. Springer, 2018.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 1999, Proceedings*, volume 1579 of *LNCS*, pages 193–207. SV, 1999.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Ce-

sare Tinelli. CVC4. In *Computer Aided Verification, CAV 2011, Proceedings*, volume 6806 of *LNCS*, pages 171 – 177. Springer, 2011.

[BCH+] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96)*.

[BDLF+16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal verification of smart contracts: Short paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016*, pages 91–96, 2016.

[BdR+10] Clark Barrett, Leonardo de Moura, Silvio Ranise, Aaron Stump, and Cesare Tinelli. The SMT-LIB initiative and the rise of SMT (HVC 2010 award talk). In *Hardware and Software: Verification and Testing, HVC 2010, Proceedings*, number 6504 in LNCS, page 3. SV, 2010.

[BG87] Andreas Blass and Yuri Gurevich. Existential fixed-point logic. In *Computation Theory and Logic, In Memory of Dieter Rödding*, volume 270 of *LNCS*, pages 20–36. SV, 1987.

[BGMR15] Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51, 2015.

[BHKS19] Martin Blicha, Antti E. J. Hyvärinen, Jan Kofroň, and Natasha Sharygina. Decomposing Farkas interpolants. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019, Proceedings*, volume 11427 of *LNCS*, pages 3–20. SV, 2019.

[BHMS20] Martin Blicha, Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. A cooperative parallelization approach for property-directed k-induction. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2020, Proceedings*, volume 11990 of *LNCS*, pages 270–292. Springer, 2020.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2011, Proceedings*, pages 70–87, 2011.

[BS96] Max Böhm and Ewald Speckenmeyer. A fast parallel SAT-solver: Efficient workload balancing. 17(4–3):381–400, 1996.

[BSS15] Tomas Balyo, Peter Sanders, and Carsten Sinz. HordeSat: A massively parallel portfolio SAT solver. In *Theory and Applications of Satisfiability Testing, SAT 2015, Proceedings*, volume 9340 of *LNCS*, pages 156–172. SV, 2015.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[CE81] E.M. Clarke and A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, 1981.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *Computer Aided Verification, CAV 2012, Proceedings*, pages 277–293, 2012.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV 2000, Proceedings*, pages 154–169, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2013, Proceedings*, volume 7795 of *LNCS*, pages 93 – 107. Springer, 2013.

[CH00] Manuel Carro and Manuel V. Hermenegildo. Tools for search-tree visualisation: The APT tool. In *Analysis and Visualization Tools for Constraint Programming, Constrain Debugging (DiSCiPl project)*, volume 1870 of *LNCS*, pages 237–252. SV, 2000.

[CK16] Sagar Chaki and Derrick Karimi. Model checking with multi-threaded IC3 portfolios. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2016, Proceedings*, pages 517–535, 2016.

[CLLZ17] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017*, pages 442–446. IEEE Computer Society, 2017.

[CLW+17] Ting Chen, Xiaoqi Li, Ying Wang, Jiachi Chen, Zihao Li, Xiapu Luo, Man Ho Au, and Xiaosong Zhang. An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks. In *Information Security Practice and Experience - 13th International Conference, ISPEC 2017, Proceedings*, pages 3–24. Springer, 2017.

[Con18] ConsenSys. Mythril, 2018. github.com/ConsenSys/mythril.

[Cra57] W. Craig. Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *The Journal of Symbolic Logic*, 22(3):269–285, 1957.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer Aided Verification, CAV 2006, Proceedings*, volume 4144 of *LNCS*, pages 81–94. SV, 2006.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2008, Proceedings*, volume 4963 of *LNCS*, pages 337 – 340. Springer, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, 2005.

[Dut14] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification, CAV 2014, Proceedings*, volume 8599 of *LNCS*, pages 737 – 744. Springer, 2014.

[ELHL20] Denis Erfurt, Martin Lundfall, Everett Hildenbrandt, and Lev Livnev. Klab, 2020. https://github.com/dapphub/klab.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2011, Proceedings*, pages 125–134, 2011.

[Eth18a] Ethereum Foundation. Ethereum: A secure decentralised generalised transaction ledger, 2018. ethereum.github.io/yellowpaper/paper.pdf.

[Eth18b] Ethereum Foundation. Solidity compiler, 2018.

[eth20a] Etherscan, 2020. https://etherscan.io.

[Eth20b] Ethereum Foundation. HEVM Ethereum evaluator, 2020. https://github.com/dapphub/dapptools/tree/master/src/hevm.

[FDHS15] Grigory Fedyukovich, Andrea Callia D'Iddio, Antti E. J. Hyvärinen, and Natasha Sharygina. Symbolic detection of assertion dependencies for bounded model checking. In *Fundamental Approaches to Software Engineering, FASE 2015, Proceedings*, volume 9033 of *LNCS*, pages 186–201. SV, 2015.

[FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A Static Analysis Framework For Smart Contracts. *arXiv e-prints*, page arXiv:1908.09878, Aug 2019.

[Fra18a] K Framework. Solidity semantics, 2018. https://github.com/kframework/solidity-semantics.

[Fra18b] K Framework. Vyper semantics, 2018. https://github.com/kframework/vyper-semantics.

[GI15] Arie Gurfinkel and Alexander Ivrii. Pushing to the top. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2015, Proceedings*, pages 65–72, 2015.

[GI17] Arie Gurfinkel and Alexander Ivrii. K-induction without unrolling. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2017, Proceedings*, pages 148–155. IEEE, 2017.

[GKJ+18] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: surviving out-of-gas conditions in ethereum smart contracts. *Proc. ACM Program. Lang.*, 2(OOPSLA):116:1–116:27, 2018.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *Computer Aided Verification, CAV 2015, Proceedings*, pages 343–361, 2015.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, CAV 1997, Proceedings*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[GSCK00] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[GSV18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *Automated Technology for Verification and Analysis, ATVA 2018, Proceedings*, pages 248–266, 2018.

[HB12] Krystof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *Theory and Applications of Satisfiability Testing, SAT 2012, Proceedings*, pages 157–171, 2012.

[HJ19] Ákos Hajdu and Dejan Jovanovic. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments, VSTTE 2019, Proceedings*, volume 12031 of *LNCS*, pages 161–179. Springer, 2019.

[HJN06] Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. A distribution method for solving SAT in grids. In *Theory and Applications of Satisfiability Testing, SAT 2006, Proceedings*, volume 4121 of *LNCS*, pages 430–435. SV, 2006.

[HJN09] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Incorporating clause learning in grid-based randomized SAT solving. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6(4):223–244, 2009.

[HJN10] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning SAT instances for distributed solving. In *Logic for Programming, Artificial Intelligence and Reasoning, LPAR-17, Proceedings*, pages 372–386, 2010.

[HJN11] Antti Eero Johannes Hyvärinen, Tommi A. Junttila, and Ilkka Niemelä. Partitioning search spaces of a randomized search. 107(2-3):289–311, 2011.

[HJS09] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 6(4):245 – 262, 2009.

[HKWB11] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing, HVC 2011, Proceedings*, pages 50–65, 2011.

[HM12] Antti Eero Johannes Hyvärinen and Norbert Manthey. Designing scalable parallel SAT solvers. In *Theory and Applications of Satisfiability Testing, SAT 2012, Proceedings*, volume 7317 of *LNCS*, pages 214–227. SV, 2012.

[HMAS16] Antti E. J. Hyvärinen, Matteo Marescotti, Leonardo Alt, and Natasha Sharygina. OpenSMT2: An SMT solver for multi-core and cloud computing. In *Theory and Applications of Satisfiability Testing, SAT 2016, Proceedings*, number 9710 in LNCS, pages 547 – 553. SV, 2016.

[HMS15] Antti E. J. Hyvärinen, Matteo Marescotti, and Natasha Sharygina. Search-space partitioning for parallelizing SMT solvers. In *Theory and Applications of Satisfiability Testing, SAT 2015, Proceedings*, volume 9340 of *LNCS*, pages 369–386. SV, 2015.

[HMS+18] Antti Hyvärinen, Matteo Marescotti, Parvin Sadigova, Hana Chockler, and Natasha Sharygina. Lookahead-based smt solving. In *Logic for Programming, Artificial Intelligence and Reasoning, LPAR-22, Proceedings*, volume 57 of *EPiC Series in Computing*, pages 418–434. EasyChair, 2018.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hol16] Gerard J. Holzmann. Cloud-based verification of concurrent software. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2016, Proceedings*, pages 311–327, 2016.

[HSR+18] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Proceedings*, pages 204–217, 2018.

[HvM09] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 155–184. IOS Press, 2009.

[HW18] Antti E. J. Hyvärinen and Christoph M. Wintersteiger. Parallel satisfiability modulo theories. In *Handbook of Parallel Constraint Reasoning.*, pages 141–178. SV, 2018.

[Hyv11] Antti E. J. Hyvärinen. *Grid-Based Propositional Satisfiability Solving*. PhD thesis, Aalto University School of Science, Aalto Print, Helsinki, Finland, 11 2011.

[JD16] Dejan Jovanovic and Bruno Dutertre. Property-directed *k*-induction. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2016, Proceedings*, pages 85–92. IEEE, 2016.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[KGDS18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: analyzing safety of smart contracts. In *Network and Distributed System Security Symposium, NDSS 2018, Proceedings*. The Internet Society, 2018.

[KS13] Arne König and Torsten Schaub. Monitoring and visualizing answer set solving. *TPLP*, 13(4-5-Online-Supplement), 2013.

[KSSS13] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of SAT solvers. In *Conference on Artificial Intelligence, AAAI 2013, Proceedings*. AAAI Press, 2013.

[KT11] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. *Electronic Proceedings in Theoretical Computer Science*, 72:55–62, 2011.

[LA03] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), Proceedings*, CGO '04, pages 75–. IEEE Computer Society, 2003.

[LCO+16] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269. ACM, 2016.

[LCWD18] Shuvendu K. Lahiri, Shuo Chen, Yuepeng Wang, and Isil Dillig. Formal specification and verification of smart contracts for azure blockchain. *CoRR*, abs/1812.08829, 2018.

[MBH+18] Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. Computing exact worst-case gas consumption for smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Proceedings*, volume 11247 of *LNCS*, pages 450–465. Springer, 2018.

[McM05] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, Proceedings*, pages 1–12, 2005.

[MGHS17] Matteo Marescotti, Arie Gurfinkel, Antti E. J. Hyvärinen, and Natasha Sharygina. Designing parallel pdr. In *International Conference on Formal Methods in Computer-Aided Design, FMCAD 2017, Proceedings*, pages 156–163, 2017.

[MHS16] Matteo Marescotti, Antti E. J. Hyvärinen, and Natasha Sharygina. Clause sharing and partitioning for cloud-based SMT solving. In *Automated Technology for Verification and Analysis, ATVA 2016, Proceedings*, pages 428–443, 2016.

[MHS18] Matteo Marescotti, Antti Hyvärinen, and Natasha Sharygina. Smts: Distributed, visualized constraint solving. In *Logic for Programming, Artificial Intelligence and Reasoning, LPAR-22, Proceedings*, volume 57 of *EPiC Series in Computing*, pages 534–542. EasyChair, 2018.

[MMH+19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *CoRR*, abs/1907.03890, 2019.

[MML12] Ruben Martins, Vasco M. Manquinho, and Inês Lynce. An overview of parallel SAT solving. 17(3):304–347, 2012.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001*, pages 530–535, 2001.

[MOA+20] Matteo Marescotti, Rodrigo Otoni, Leonardo Alt, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. Accurate smart contract verification through direct modelling. In *Proc. ISoLA 2020, to appear*, LNCS. Springer, 2020.

[MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. 48(5):506–521, 1999.

[NKS+18] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. *CoRR*, abs/1802.06038, 2018.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937 – 977, 2006.

[par17] Parity security alert, 2017. https://www.parity.io/security-alert-2/.

[PC13] Hristina Palikareva and Cristian Cadar. Multi-solver support in symbolic execution. In *Computer Aided Verification, CAV 2013, Proceedings*, volume 8044 of *LNCS*, pages 53–68. SV, 2013.

[PDT+20] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. VerX: safety verification of smart contracts. In *IEEE Symposium on Security and Privacy (S&P), IEEE SP 2020, Proceedings*, 2020.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, 1982.

[Rei14] Christian Reisenberger. PBoolector: a parallel SMT solver for QF_BV by combining bit-blasting with look-ahead. Master's thesis, Johannes Kepler Univesität Linz, Linz, Austria, 2014.

[RS10] Grigore Rosu and Traian Florin Serbanuta. An Overview of the K Semantic Framework. *The Journal of Logic and Algebraic Programming*, 79(6):397 – 434, 2010.

[RSMO15] Emil Rakadjiev, Taku Shimosawa, Hiroshi Mine, and Satoshi Oshima. Parallel SMT solving and concurrent symbolic execution. In *2015 IEEE TrustCom/BigDataSE/ISPA, Proceedings*, pages 17–26, 2015.

[Sim00] Patrick Simons. *Extending and Implementing the Stabel Model Semantics*. PhD thesis, Helsinki University of Technology, 2000.

[Sin07] Carsten Sinz. Visualizing SAT instances and runs of the DPLL algorithm. *J. Autom. Reasoning*, 39(2):219–243, 2007.

[smt20] Smtchecker documentation, 2020. https://solidity.readthedocs.io/en/v0.6.6/security-considerations.html#formal-verification.

[sol20] Solidity documentation, 2020. https://solidity.readthedocs.io.

[TDDC+18] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. ACM, 2018.

[the20] theDAO, 2020. https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413.

[Tin02] Cesare Tinelli. A dpll-based calculus for ground satisfiability modulo theories. In *Logics in Artificial Intelligence, European Conference, JELIA 2002, Proceedings*, pages 308–319, 2002.

[vyp20] Vyper documentation, 2020. https://vyper.readthedocs.io.

[WHdM09] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. A concurrent portfolio approach to SMT solving. In *Computer Aided Verification, CAV 2009, Proceedings*, volume 5643 of *LNCS*, pages 715–720. SV, 2009.

[Why18] Why3. Why3, 2018. http://why3.lri.fr.

[ZBH96] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: A distributed propositional prover and its application to quasigroup problems. 21(4):543–560, 1996.

[ZM88] Ramin Zabih and David McAllester. A rearrangement search strategy for determinig propositional satisfiability. In *Conference on Artificial Intelligence, AAAI 1988, Proceedings*, pages 155–160, 1988.