Automated Verification of Blockchain Technologies with Correctness Guarantees

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
Rodrigo Benedito Otoni

under the supervision of
Patrick Eugster and Natasha Sharygina

October 2023

Dissertation Committee

Gabriele Bavota	Università della Svizzera italiana, Lugano, Switzerland
Arie Gurfinkel	University of Waterloo, Waterloo, Canada
Shuvendu Lahiri	Microsoft Research, Redmond, USA
Fernando Pedone	Università della Svizzera italiana, Lugano, Switzerland

Dissertation accepted on 9 October 2023

Research Advisor	Co-Advisor
Patrick Eugster	**Natasha Sharygina**

PhD Program Director

Walter Binder / Stefan Wolf

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

Rodrigo Benedito Otoni
Lugano, 9 October 2023

Abstract

Blockchain technologies have drawn significant attention from both academia and industry over the last decade, with increasing adoption by the general public and potential to drastically change the way in which individuals and institutions interact with each other. Due to their extensive use as a means to hold and manipulate financial assets, they are likely targets of attacks, with assets estimated in the order of millions of US Dollars having already been lost in the past. In light of this, the ability to ensure the absence of vulnerabilities is of crucial importance.

This work addresses the need for automated verification of blockchain technologies with correctness guarantees. The blockchain space is approached both at the platform and the application level, with the use of verification techniques based on first-order logic being investigated as a means to tackle the growing need of assurances in this domain. To strengthen the guarantees provided, the use of correctness witnesses to validate the results of logic solvers dedicated to logic fragments of interest is also investigated.

This dissertation draws on recent advances in the field of symbolic model checking, specifically on techniques based on satisfiability modulo theories (SMT) and constrained Horn clauses (CHC), and extend the state of the art in a fourfold manner. It (i) proposes a novel scalable SMT-based model checking approach for distributed algorithms, which are the cornerstone of blockchain platforms, specified in TLA$^+$. It (ii) evaluates and enhances a direct modelling CHC-based model checking approach for smart contracts, which are programs executing on top of blockchain platforms, written in Solidity. It (iii) proposes a novel format for witnesses of SMT unsatisfiability results, which leads to witnesses that are compact and lightweight to produce and validate. It (iv) proposes a novel proof-backed approach for the validation of CHC satisfiability results.

The first two contributions are directly applicable in the blockchain domain and they interface with the last two via their use of SMT and CHC. Despite the focus on one specific domain, the insights presented in this dissertation can in principle be applied in the general case and serve as a basis for further research.

Acknowledgements

I would like to thank all those who supported me during the long journey that led to this dissertation. I have grown immensely, both personally and professionally, through the years of my PhD studies and I am deeply grateful for the help given.

I am very thankful to my advisors, Prof. Patrick Eugster and Prof. Natasha Sharygina, for the opportunity to work with them and for their expert guidance, which opened my eyes to so many things and led me to fascinating discoveries.

My time at the Università della Svizzera italiana was a wonderful one, and for that I thank all the faculty members and colleagues with whom I had the pleasure to work. In special, I thank Dr. Antti Hyvärinen, for his kind and insightful advice on many topics, as well as Dr. Anita Buckley, Dr. Martin Blicha, Dr. Matteo Marescotti, Shamiek Mangipudi, Konstantin Britikov, Davide Rovelli, Dr. Sepideh Asadi, Dr. Pavel Chuprikov, Dr. Pierre-Louis Roman, Masoud Asadzadeh, and Gerald Prendi, for all the enjoyable discussions.

During my PhD I had the opportunity to do an internship at Informal Systems, which proved to be a great experience. I thank all the company members that I was fortunate enough to interact with for their kindness and willingness to help me. In special, I thank Dr. Igor Konnov, for his mentoring and unending positivity, as well as Dr. Jure Kukovec, Shon Feder, Gabriela Moreira, and Dr. Thomas Pani, for welcoming me into their team with open arms.

For providing the financial support that enabled my studies, I gratefully thank both the European Research Council (grant FP7-617805) and the Swiss National Science Foundation (grant 2000021_197353).

I have always been blessed with great mentors. For instilling in me the love for research that led me here, I thank my BSc and MSc advisors, Prof. Leila Silva, Prof. Ana Cavalcanti, and Prof. Augusto Sampaio, to whom I'll be forever grateful.

Moving to Switzerland was a big change for me, which was made much easier by the helping hand of my cousins Nicole and Priscila, and of Severin Hirt, whose assistance was greatly appreciated.

Lastly, I thank my parents for the unconditional support they have always given me and for always driving me to do my best in all circumstances.

Contents

Chapter 1

Introduction

Distributed ledgers, which underpin the blockchain technologies, allow secure transactions between distrusting parties to take place without the need of a supervising authority, such as a bank. Their rise promises to disrupt the established way in which individuals and institutions interact with each other, projecting to providing a cheaper and more secure way to exchange goods and services.

The use of distributed ledgers was initially focused on the creation of fungible tokens in the form of cryptocurrencies, e.g., Bitcoin [Nakamoto, 2008]. As interest in the area grew, blockchain platforms started to also support programs capable of automatically executing contractual agreements written as code, in the form of smart contracts. This set the stage for the rise of a whole new ecosystem, one in which developers make applications via smart contracts and rely on blockchains to securely transmit tokens as intended.

Due to their extensive use as a means to hold and manipulate financial assets, blockchain technologies are likely targets of attacks. Such attacks can attempt to exploit vulnerabilities in smart contracts applications [Atzei et al., 2017] or in the blockchain platforms in which they rely upon [Chen et al., 2020], with assets estimated in the order of millions of US Dollars having already been lost in the past. The ability to ensure that no vulnerabilities are present in both smart contract applications and blockchain platforms is thus of paramount importance, with formal verification being an ideal approach to achieve this goal.

Formal verification is a broad term referring to a number of logic-based techniques that aim at mathematically ensuring that a program or system behaves according to some specification, which can range from the simple absence of overflows to complex functional properties. The different formal verification techniques vary in their level of automation, expressiveness, and efficiency, often being best suited to specific settings. One technique that has seen significant

advances in the last two decades is the encoding of verification tasks in, and the solving of, first-order logic (FOL) formulas [Kroening and Strichman, 2016]. Many logic solvers have been developed and used in both academia and industry, due to their combination of push-button automation, expressive formalisms, and high level of efficiency in many real world scenarios.

Despite their extensive usage in verification, with one explicit goal being to prevent bugs that can be exploited by attackers, logic solvers are themselves far from being immune to bugs. This has been made evident by the constant uncovering of issues during tool competitions over the last few years, with many state-of-the-art solvers yielding contradictory results [Barbosa et al., 2022b; de Angelis and Govind V. K., 2022]. This is a critical problem, since unsound results can potentially have catastrophic repercussions, which makes correctness guarantees vital to practical solver usage.

This dissertation focuses on automated verification of blockchain technologies via logic solvers with correctness guarantees. The sections of this chapter provide an overview of the relevant blockchain technologies (Section 1.1), automated verification techniques (Section 1.2), and logic solvers' correctness guarantees (Section 1.3), as well a summary of the challenges and contributions (Section 1.4), and an outline of the remainder of the manuscript (Section 1.5).

1.1 Blockchain Technologies

Blockchain platforms are, at their core, a large number of geographically distributed replicated state machines. Each replica holds a copy of a ledger, with a distributed algorithm, often called a protocol, ensuring that ledger updates happen in a consistent way. Such algorithms are often extremely complex, since they have to ensure consensus in the presence of Byzantine faults, i.e., they have to ensure that faulty nodes behaving arbitrarily do not impede other nodes from functioning correctly [van Steen and Tanenbaum, 2017]. Many different blockchain platforms exist, each with their own particular protocol [Cachin and Vukolic, 2017]. While initially used only for direct token storage and transfer, the advent of smart contracts allowed blockchains to serve as the basis for a wide variety of new applications.

Smart contracts are distributed programs designed to manage and enforce contract transactions without relying on trusted authorities, but instead exploiting blockchain platforms to achieve this goal. A prime example of such a platform is Ethereum [Buterin, 2014], one of the most used blockchain platforms and the main target for the development of smart contracts. In the specific case of

Ethereum, smart contracts are commonly implemented in high-level languages such as Solidity and Vyper, and then compiled to low-level Ethereum virtual machine (EVM) bytecode [Wood, 2015], which is deployed on the blockchain itself. Despite having many elements in common with general programs, smart contracts have their own specificities that make developing and reasoning about them to be quite different. A good example of such differences is the transactional nature of smart contracts, which ensures that function executions either terminate successfully or have all their changes reversed, due to how the effects of these executions are stored in the blockchain. Besides their immediate application in the financial sector [Egelund-Müller et al., 2017; Rius and Gashier, 2020], smart contracts have also been used in a variety of other areas, such as healthcare [Gordon and Catalini, 2018; Zhang et al., 2018], energy management [Andoni et al., 2019; Wang et al., 2019], and gaming [Min et al., 2019], among others [Rouhani and Deters, 2019].

1.2 Automated Verification

Many techniques fall under the formal verification umbrella, including model checking [Clarke et al., 2018], static analysis [Rival and Yi, 2020], and theorem proving [Bertot and Castéran, 2004], each with their own level of automation and list of strengths and weaknesses. Of those, model checking is one technique that has seen many successes over the years, recognised most notably by the 2007 Turing Award. In addition to being the target of academic interest, model checking has increasingly been used in industry [Woodcock et al., 2009], with one of the most famous examples of this trend being its adoption by Amazon Web Services [Newcombe et al., 2015].

In essence, model checking consists of encoding a program or system, together with a desired property, in a suitable mathematical formalism, and then checking if all behaviours respect the given property. The choice of formalism, e.g., state transition graphs checked via graph traversal, has a significant impact on model checking performance and is thus a major focus of research. Despite successes, model checking remains a daunting task in practice, due to limited scalability caused by the well known state explosion problem [Clarke et al., 2012]. This task, already highly nontrivial in cases consisting exclusively of sequential behaviours, is made much harder in a distributed setting, in which communication among different nodes is extensive and done over potentially unreliable channels, with the possibility of node failures further complicating reasoning. In an effort to better address this issue, model checking approaches

based on symbolic reasoning via logic formulas have been proposed, leading to improvements not only in the verification of programs, e.g., written in languages such as C/C++ [Gurfinkel et al., 2015] and Java [Kahsai et al., 2016], but also also of system's specifications, e.g., written in languages such as TLA$^+$ [Konnov et al., 2019] and Ivy [McMillan and Padon, 2020].

Symbolic reasoning is commonly done in fragments of FOL, since they are capable of representing many interesting verification problems, with the choice of fragment being based on a trade-off between expressiveness and efficiency. Automated reasoning of FOL formulas is done via logic solvers, with each such solver catering to one or more FOL fragments. The most well known solver categories are arguably those of Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) [Kroening and Strichman, 2016], which respectively cater to formulas in the propositional fragment of FOL and in extensions of it with theories such as arithmetics, arrays, and bit vectors. Another category of interest is that of constrained Horn clauses (CHC) [Gurfinkel and Bjørner, 2019], which has been shown to be a match for Hoare logic [Hoare, 1969] with many practical uses [Bjørner et al., 2015].

1.3 Correctness Guarantees

Logic solvers are a central part of many modern verification tools, functioning as their back-end reasoning engines. Despite their importance, these solvers are known to contain bugs, as exemplified by the 2022 edition of the annual SMT competition, in which 18 benchmarks led to at least two state-of-the-art solvers disagreeing on the results [Barbosa et al., 2022b]. In light of this, having guarantees about solvers' results is of paramount importance. One approach to achieve this goal is to formally verify the solvers' code, as has been done for read-eval-print loop (REPL) [Kumar et al., 2014] and garbage collector [Sandberg Ericsson et al., 2019] implementations. In spite of the strong guarantees provided, this approach incurs a high cost to verify the existing codebase and any future modifications to it, as well as potentially preventing many code optimizations to be made, which are essential for solver performance. Another, less invasive, approach, is to validate solvers' outputs, rather than verifying the solvers themselves. This requires a solver, in addition to producing its standard output, to also produce a witness that can be used by an independent tool to validate the given result. Currently, the community is moving towards the second approach, with many witness formats being proposed to validate the outputs of SAT [Heule et al., 2013a; Cruz-Filipe et al., 2017; Baek et al., 2021] and SMT [Stump et al., 2013;

Schurr et al., 2021; Hoenicke and Schindler, 2022] solvers, and both the annual SAT and SMT competitions now following this approach. Codebase verification can still be applied, however, targeting instead the validation tools [Heule et al., 2017; Lammich, 2020], which are much less complex and easier to maintain.

1.4 Challenges and Contributions

Driven by the critical need to prevent vulnerabilities that can be exploited by attackers, significant research effort was made in recent years in order to provide practitioners with the tooling necessary to formally verify many aspects of blockchain platforms and smart contracts. The same can be said of the need to provide correctness guarantees of logic solvers' results, since a large number of modern verification tools, targeting the blockchain space or otherwise, rely on logic solvers. Despite advances, however, many challenges still remain, of which four are considered in this dissertation. First, verifying real world distributed systems is often infeasible with existing approaches, since currently available approaches struggle to cope with the sheer complexity of the task. Second, the execution model of smart contracts is quite different from those of general programs, requiring its specificities to be accurately handled during verification. Third, witnesses produced to validate logic solvers' results tend to be excessively large, up to the order of gigabytes, which is a limiting factor w.r.t. their practical usage. Fourth, witness validation approaches for certain FOL fragments require complex procedures that may themselves have invalid results, undermining any potential guarantees given.

The contributions of this dissertation, summarised in Figure 1.1, address the challenges just described and can be split into two categories: automated verification and correctness guarantees. Challenges one and two fall under automated verification and are tackled via novel model checking approaches. To increase efficiency of verification of distributed systems a scalable SMT-based model checking approach of TLA$^+$ specifications was developed, described in Section 1.4.1. TLA$^+$ is a language widely used to formalise algorithms underpinning distributed systems, being thus an ideal target to achieve this goal. To aid in precisely representing smart contracts specificities during verification a large scale evaluation of a CHC-based model checking approach of Solidity programs was conducted and an extension to it proposed, described in Section 1.4.2. Solidity is by far the most used language for the implementation of smart contracts, being thus a prime target for verification. Challenges three and four fall under correctness guarantees and are tackled via novel witness production and checking approaches. To ad-

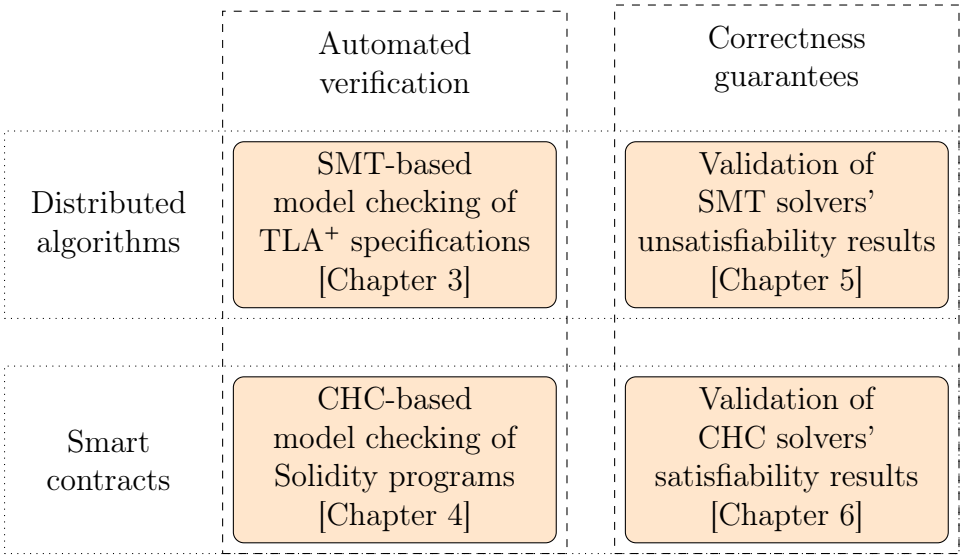


Figure 1.1. Contributions of this dissertation and how they are organised.

dress the issue of witnesses sizes in the context of SMT solving a novel format for unsatisfiability witnesses with compactness as its main priority was developed, described in Section 1.4.3. To strengthen the validation guarantees achieved in the context of CHC solving a novel proof-backed approach for the validation of satisfiability witnesses was developed, described in Section 1.4.4.

1.4.1 Model Checking of TLA$^+$ Specifications

Reasoning about distributed algorithms is a highly nontrivial task that can greatly benefit from tool support. Verification tooling can provide not only automation, but also guarantees regarding the results, which are much needed since pen-and-paper proofs can contain mistakes that go unnoticed for years [Lincoln and Rushby, 1993]. TLA$^+$ is a specification language based on the temporal logic of actions (TLA) which allows users to describe the expected behaviour of a system while abstracting away implementation details that do not impact high-level properties, such as memory management [Lamport, 2002]. It is widely used to formalise and reason about distributed algorithms, both in academia and in industry [Newcombe et al., 2015], with a handful of tools being available to aid in verification tasks [Konnov et al., 2022].

Symbolic model checking of TLA$^+$ specifications was spearheaded by Konnov et al. [2019], who proposed an encoding of TLA$^+$ into SMT constraints over uninterpreted constants and arithmetics. Their tool, APALACHE, saw many successes,

but encountered difficulties when handling particularly complex specifications. One key aspect of APALACHE's approach is the encoding of TLA$^+$ data structures as uninterpreted constants, which prevents the structural information present in the input specification to be reflected in the SMT formula encoding it, leading to a negative impact on performance.

The structural information not forwarded to the SMT solver has the potential to significantly improve solving efficiency, which is the determining factor in overall model checking performance. In light of this, an alternative SMT encoding was proposed that makes full use of the SMT theory of arrays [de Moura and Bjørner, 2009] to encode the main TLA$^+$ data structures. The proposed encoding was implemented in APALACHE and compared against APALACHE's original encoding and the explicit state enumeration approach followed by the model checker TLC [Yu et al., 1999]. The evaluation performed indicates that embedding structural information into the SMT formulas has a significant positive impact on performance, with this insight potentially generalising to other contexts.

The results of this work have been published at TACAS'23 [Otoni et al., 2023b] and are presented in detail in Chapter 3.

1.4.2 Model Checking of Solidity Programs

Due to their deployment on blockchain platforms, smart contracts have a number of specificities not found in general programs. Execution-wise, their transactional nature, in which a function execution either finishes successfully or has all its changes reversed, is the main difference in relation to traditional programs, such as those written in C/C++ and Java. Regarding the code itself, it is immutable after deployment, which prevents vulnerability fixes, publicly visible, allowing potential attackers to search for code exploits, and freely available, meaning that any user can interact with its interface. The combination of these features makes smart contract verification essential, with many approaches aiming to aid in this being proposed over the years, based on model checking [Kalra et al., 2018; Albert et al., 2019; Wang et al., 2020], static analysis [Luu et al., 2016; Mossberg et al., 2019; Permenev et al., 2020], or other techniques [Bhargavan et al., 2016; Tsankov et al., 2018; Hajdu and Jovanovic, 2020].

A trend among verification approaches targeting smart contracts is the attempt to reuse existing tools, designed to reason about general programs, to reason about contracts' behaviours. The clear benefit of this is the reuse of established off-the-shelf tools, which can provide much desired stability and efficiency, but an important drawback is the need of a translation from the domain of smart contracts to the tool domain, which adds a new unnecessary layer in the verifica-

tion framework that is error-prone to develop, requires correctness proofs of its own, and can negatively impact precision and efficiency. The attempt to reuse existing tools is, thus, interesting at first, but not an ideal lasting solution, with the alternative being the development of purpose-built algorithms and tooling to handle all features present in the smart contracts domain natively.

To help tackle this challenge the CHC-based model checking approach targeting Solidity programs by Marescotti et al. [2020] was assessed and extended. The approach is based on direct modelling, meaning that verification conditions are generated in the target formalism directly from the control-flow graph (CFG) of the contract, using domain-specific knowledge, and without any intermediary steps. Solidity contract verification using constrained Horn clauses, SOLICITOUS for short, implements the approach and acts as the CHC model checking engine of SOLCMC [Alt et al., 2022], being capable of checking code assertions as well as predetermined properties, such as overflows.

The evaluation of SOLICITOUS was done by executing it with 22446 real world Solidity contracts currently deployed on the Ethereum blockchain and comparing it against three publicly available tools suitable for automated verification of Solidity assertions, namely SRI's SOLC-VERIFY [Hajdu and Jovanovic, 2020], Microsoft's VERISOL [Wang et al., 2020], and ConsenSys' MYTHRIL [ConsenSys, 2021]. Quantitative and qualitative analyses of the results obtained were performed, including a manual inspection of all bugs found. In addition, an extension of the encoding to gather the exact order and arguments of function calls for better counterexample (CEX) generation was proposed.

The results of this work have been published at TOPS [Otoni et al., 2023c] and are presented in detail in Chapter 4.

1.4.3 Validation of SMT Solvers' Unsatisfiability Results

A SMT formula can be either satisfiable, SAT for short (not to be confused with Boolean satisfiability), meaning that there exists an assignment to its variables such that the formula evaluates to true, or unsatisfiable, UNSAT for short, if no such assignment is possible. Witnesses for satisfiable results are called SAT models, while witnesses for unsatisfiable results are called UNSAT proofs. The emphasis in this dissertation is on UNSAT proofs for quantifier-free fragments of FOL, in particular showing to be correct the unsatisfiability of problems in NP, as this is more challenging than satisfiability in NP, assuming NP $\neq$ coNP.

The production of UNSAT proofs can currently be done by a number of SMT solvers, each using their own specific proof format [Barrett et al., 2015]. Proof checking was initially done by replaying the proof inside theorem provers such

as Coq [Armand et al., 2011; Ekici et al., 2017] and Isabelle/HOL [Böhme and Weber, 2010; Blanchette et al., 2016]. Formats capable of being checked by lightweight tools, such as the one based on the logical framework with side conditions (LFSC) [Stump et al., 2013], also emerged, providing a way to validate SMT solvers' UNSAT results without relying on the infrastructure of a theorem prover. These formats, however, tend to produce witnesses that are quite large, hindering their integration into SMT-based tooling.

To address this issue a novel proof format with the goal of producing compact proofs was proposed. The proofs in this format start from a directed acyclic graph (DAG) representation of the input formula and end, essentially, in an empty resolvent of a resolution refutation. The DRAT format by Wetzler et al. [2014] was used as a base, to reason about the propositional fragment of FOL, and extend with real arithmetic certificates based on Farkas' Lemma, lemmas for simple forms of Gomory cuts, a natural formalization of equality of functions, and an axiomatization of the Tseitin transformation and the De Morgan rules. The focus was on the theories of quantifier-free linear real arithmetic (QF_LRA), quantifier-free linear integer arithmetic (QF_LIA), and quantifier-free uninterpreted functions (QF_UF), as these underpin most other SMT theories. The SMT solver OpenSMT [Bruttomesso et al., 2010] was instrumented to produce proofs in the proposed format and the Theory-Specific Witness Checker, TSWC for short, was implemented to check them. The evaluation performed indicates that the format leads to smaller proofs that can be produced with a low overhead during solving, relative to other proof producing solvers, and can be efficiently checked.

The results of this work have been published at DAC'21 [Otoni et al., 2021] and are presented in detail in Chapter 5.

1.4.4 Validation of CHC Solvers' Satisfiability Results

The input of a CHC solver is a conjunction of logical implications containing uninterpreted predicates, with the task of the solver being to decide if *false* can be derived or not. If it can the input is considered unsatisfiable, and if it cannot the input is considered satisfiable. As with SMT solving, the SAT and UNSAT acronyms are used as shorthands for satisfiable and unsatisfiable, and witnesses for these results are respectively called models and proofs. The production of witnesses is a common feature of modern CHC solvers, but efforts in witnesses validation are limited. The validation of models is done via SMT queries, and is currently supported only by an ad hoc validator tied to the SMT solver Z3 [de Moura and Bjørner, 2008b]. Unlike is the case for models, however, the proofs produced cannot be validated with available tooling, given that, to the best of our knowl-

edge, no proof checking approach currently exists.

Since CHC model validation is underpinned by SMT solving, the same concern regarding the correctness of CHC solvers results is put on the validation itself, i.e., on the correctness of SMT solvers' results. To address this, a two-layered validation approach to provide additional guarantees about the results obtained was proposed. The first layer, consisting of the SMT queries responsible for model validation, is enhanced by a second layer, consisting of the production and checking of SMT proofs, with the result obtained being forwarded to the user or tool interacting with the CHC solver. The approach is generic w.r.t. FOL theories and solvers, and is also very modular, enabling different SMT solvers to be used in the validation, further increasing assurances.

In order to both make the approach practical and conduct an evaluation, the modulAr consTrained Horn clauses modEl validatioN frAmework, ATHENA for short, was developed, capable of catering to different combinations of state-of-the-art CHC and SMT solvers. Concretely, the framework was used to validate the models produced by three CHC solvers, with each model produced being separately validated by five proof producing SMT solvers. In addition, all the proofs produced in the proof formats currently supported by automated proof checkers were checked. All 955 linear integer arithmetic (LIA) benchmarks from the 2022 edition of the annual CHC competition were used in the evaluation, 499 containing only linear Horn clauses, i.e., implications with a single uninterpreted predicate in the implicant, and 456 containing nonlinear Horn clauses, i.e., implications with multiple uninterpreted predicates in the implicant. The results indicate that model validation can be used in practice, with the majority of the models being validated with available tooling, but models sizes are a concern.

The results of this work have been accepted to iFM'23 [Otoni et al., 2023a] and are presented in detail in Chapter 6.

1.5 Outline

The remainder of the dissertation is structured as follows. The necessary background is covered in Chapter 2. The proposed SMT-based model checking approach targetting TLA$^+$ specifications is presented in Chapter 3. The evaluation and extension of the CHC-based model checking approach targeting Solidity programs by Marescotti et al. is presented in Chapter 4. The proposed approach to validate SMT unsatisfiability results is presented in Chapter 5. The proposed approach to validate CHC satisfiability results is presented in Chapter 6. Finally, closing remarks are presented in Chapter 7.

Chapter 2

Background

This chapter introduces the basics of symbolic model checking, in Section 2.1, the TLA$^+$ language, in Section 2.2, the Solidity language, in Section 2.3, satisfiability modulo theories, in Section 2.4, and constrained Horn clauses, in Section 2.5.

2.1 Symbolic Model Checking

Symbolic model checking is a state-of-the-art approach for software verification. It is a variation of the original model checking approach, now commonly referred to as explicit state model checking. In contrast to the original approach, which represents all states explicitly, usually by means of a graph structure, symbolic model checking represents sets of states as logical constraints under a chosen formalism. This shift in representation allowed for the production of models that are significantly more compact and enabled efficient reasoning about systems that were previously intractable. Here an overview of symbolic model checking and its uses is provided, with further details being available in the handbook by Clarke et al. [2018].

Similar to its explicit state counterpart, symbolic model checking has two steps: modelling and checking. While these steps are in principle independent, the modelling formalism chosen is often strongly correlated with one or more checking algorithms. It is also worth pointing out that different encoding approaches based on the same formalism can be more or less amenable to specific checking algorithms. Expressive formalisms allow for natural and efficient encoding of systems and properties to be verified, but the higher the expressiveness the harder the problem of deciding if a given property holds or not, with some formalisms even leading to the checking step to be undecidable.

Once a system is modelled, the checking algorithm tries to establish if there

exists a counterexample (CEX) for the property being checked. A CEX is a sequence of transitions starting at the initial state of the model, i.e., the state representing the starting configuration of the system being analysed, and reaching a so called bad state, i.e., a state representing a configuration of the system being analysed that violates the property being checked; transitions in this context represent system's behaviours leading to a change in configuration, e.g., a variable update. If it can be shown that no CEX exists, then the system is considered safe, i.e., the desired property holds in all possible executions. If, instead, a CEX exists, then a bug has been found.

The first formalism used to represent a set of explicit states as a single, symbolic, state, was binary decision diagrams (BDD) [Bryant, 1986]. The use of BDD was an improvement over the state of the art [Burch et al., 1992], but the exponential increase in the number of states, known as the state explosion problem, remained an obstacle to practicality in software verification.

The next formalism to be widely used in symbolic model checking was propositional logic, guided by advances in Boolean satisfiability (SAT) solving [Biere et al., 1999]. SAT solvers are tools specifically designed to decide if there exists a satisfiable assignment to a formula in propositional logic, i.e., if there is an assignment to all free variables such that the formula evaluates to true. Satisfiability of general propositional formulas is a NP-complete problem, but despite this, solvers with clever heuristics are able to solve formulas representing many practical problems. Such solvers are commonly used to support bounded model checking (BMC), which is an approach for bounded reasoning that iteratively tries to find a CEX of an ever increasing length k for a given property. If no CEX of length k exists and there are still unexplored states, BMC either increments k or terminates, depending on how deep the user decides to explore. Given its bound, BMC is only able to establish safety for systems that can be represented by a finite amount of states.

SAT-based model checking proved to be a great advance, but propositional logic was found to be very restrictive to model software. The search for expressiveness led to the adoption of another formalism for model checking, called satisfiability modulo theories (SMT). SMT combines propositional logic with different first-order logic (FOL) theories, such as arithmetics, to provide better abstractions for modelling, and is discussed in Section 2.4. In addition to providing a more natural way of representing systems, better abstractions often lead to performance improvements in solving, which propagate to model checking as a whole. The benefits that come with using FOL are, however, matched by the drawback of dealing with formulas whose solving is, in the general case, undecidable. A common way to circumvent this issue is to restrict the modelling to

decidable fragments of FOL, which limit, for instance, the use of logic quantifiers. Similar to SAT, a common use of SMT is to support bounded model checking, with this combination being an active are of research. One of the contributions of this dissertation, the novel SMT-based encoding of TLA$^+$ described in Chapter 3, falls in this category.

One of the promises of formal verification is to not only allow for bugs to be found if they exist, but to also be able to ensure the absence of said bugs in the opposite case, i.e., establish safety. While relying on a bounded approach, all safety guarantees given are also bounded, which is an important limitation. One formalism that has gained traction over the last decade and allows for unbounded reasoning is constrained Horn clauses (CHC). The CHC formalism is discussed in Section 2.5, but the two aspects of it that are relevant here are its ability to enable unbounded model checking and the need to deal with an undecidable problem in order to do so. The common way in which solving is approached is through the incremental construction of a safe inductive invariant, i.e., an invariant that is implied by the initial state and that defines a closed set of states which contains no bad states. Despite the decidability issue, this approach has been shown to be practical in many different contexts.

One example of unbounded model checking pertinent to this dissertation is the CHC-based smart contract verification approach by Marescotti et al. [2020]. It consists of an encoding of Solidity contracts into CHC and is detailed in Section 4.1; the section also contains four rules proposed to extend the original encoding. Due to the undecidability of CHC solving, it is critical to asses how any encoding will fare in practice, with the assessment of this encoding, discussed in Section 4.3, being one the contributions of this dissertation. The evaluation performed not only indicates that this CHC encoding is practical, but also that it can outperform comparable approaches.

Many state-of-the-art logic solvers exist and are in active development. To further illustrate the interplay between model checkers and solvers, the SMT solver OpenSMT [Bruttomesso et al., 2010] serves as a good additional example. OpenSMT is a solver specifically designed to be a playground for different solving procedures and SMT-based verification approaches, which makes it an excellent tool for research activities. On the solving side, it currently supports eight different quantifier-free SMT theories and serves as the base of the CHC solver Golem [Blicha et al., 2023]. On the model checking side, it is the base of four different bounded checking tools, namely FunFrog [Sery et al., 2012], eVolCheck [Fedyukovich et al., 2013], HiFrog [Alt et al., 2017], and UpProver [Asadi et al., 2020].

2.2 The TLA$^+$ Language

TLA$^+$ is a language for modelling systems, particularly suited to describing systems dealing with distribution. The language is introduced via a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [1985], shown in Figure 2.1. Here the focus is on the most relevant TLA$^+$ constructs, with further details being available in the reference book by Lamport [2002].

The first notable aspect of TLA$^+$ is that specifications may be parametrised, e.g., the number of processes and faults may not be fixed. In the example, the keyword CONSTANTS, in line 3, is used to declare its parameters: N, the total number of processes, and T and F, the maximal and actual number of faulty processes. It is important to understand, however, that while a specification may be parametrised, model checking can only be carried out for a specific instance of the protocol at a time, e.g., $N = 4$ and $T = F = 1$. Parameter declarations are followed by variable declarations, by the use of the VARIABLES keyword, in line 4. Variables define the states of the state-machine that the specification describes, with each state being defined by the combination of the values held by each variable. In the example, each state is defined by the values of $sentEcho$, $sentReady$, $rcvdEcho$, $rcvdReady$, and pc.

The remaining TLA$^+$ operators describe state machine transitions or properties to be checked, and are defined using $\triangleq$. Two operators are of special significance, one that defines the initial state predicate and one that plays the role of the transition operator. In the example, these operators are $Init$, in line 8, and $Next$, in line 37. Concretely, $Init$ defines the starting point for state space exploration and $Next$ defines the exploration itself. Transitions are guided by constraints that must hold in both pre-transition states, represented by nonprimed variables, and post-transition states, represented by primed variables.

Specifications may optionally define invariants, i.e., properties that should hold in every reachable state. There is no special syntax for invariants, and they are provided by name to model checkers at invocation time. In the example, there is one invariant, $NoDecide$, in line 44. A specification satisfies $NoDecide$ if no state reachable from $Init$ via any number of $Next$ transitions has $pc[p] =$ "AC", for some $p \in Corr$. Abstractly, this invariant holds iff $Decide$ can never be taken.

To illustrate, consider having $N = 1$ and $T = F = 0$ in the example. In this case $Init$ would admit a state with $pc = [p \in \{1\} \mapsto$ "$V1$"$]$, $rcvdEcho = rcvdReady = [p \in \{1\} \mapsto \{\}]$, and $sentEcho = sentReady = \{\}$. Furthermore, $Next$ would allow a transition via $SendEcho$ from that state to one with $pc = [p \in \{1\} \mapsto$ "EC"$]$, $rcvdEcho = rcvdReady = [p \in \{1\} \mapsto \{\}]$, $sentEcho = \{1\}$, and $sentReady = \{\}$. Finally, a state with $pc = [p \in \{1\} \mapsto$ "cat"$]$, $rcvdEcho =$

```
1  ──────────────────────── MODULE ABA ────────────────────────
2  EXTENDS Integers, FiniteSets
3  CONSTANTS N, T, F
4  VARIABLES sentEcho, sentReady, rcvdEcho, rcvdReady, pc
5  Corr ≜ 1 .. (N − F)          The set of correct processes
6  Byz  ≜ (N − F + 1) .. N      The set of Byzantine processes
7  Proc ≜ 1 .. N                The set of all processes
8  Init ≜ ∧ pc ∈ [Corr → {"V0", "V1"}]
9         ∧ rcvdEcho  = [p ∈ Corr ↦ {}]
10        ∧ rcvdReady = [p ∈ Corr ↦ {}]
11        ∧ sentEcho  ∈ SUBSET Byz
12        ∧ sentReady ∈ SUBSET Byz
13 Receive(p, nextEcho, nextReady) ≜
14     ∧ rcvdEcho[p]  ⊆ nextEcho
15     ∧ rcvdReady[p] ⊆ nextReady
16     ∧ rcvdEcho'  = [rcvdEcho EXCEPT ![p] = nextEcho]
17     ∧ rcvdReady' = [rcvdReady EXCEPT ![p] = nextReady]
18 SendEcho(p, nextEcho, nextReady) ≜
19     ∧ ∨ pc[p] = "V1"
20       ∨ ∧ pc[p] = "V0"
21         ∧ ∨ Cardinality(nextEcho)  ≥ (N + T + 2) ÷ 2
22           ∨ Cardinality(nextReady) ≥ T + 1
23     ∧ pc' = [pc EXCEPT ![p] = "EC"]
24     ∧ sentEcho' = sentEcho ∪ {p}
25     ∧ UNCHANGED sentReady
26 SendReady(p, nextEcho, nextReady) ≜
27     ∧ pc[p] = "EC"
28     ∧ ∨ Cardinality(nextEcho)  ≥ (N + T + 2) ÷ 2
29       ∨ Cardinality(nextReady) ≥ T + 1
30     ∧ pc' = [pc EXCEPT ![p] = "RD"]
31     ∧ sentReady' = sentReady ∪ {p}
32     ∧ UNCHANGED sentEcho
33 Decide(p, nextReady) ≜
34     ∧ pc[p] = "RD" ∧ Cardinality(nextReady) ≥ 2 * T + 1
35     ∧ pc' = [pc EXCEPT ![p] = "AC"]
36     ∧ UNCHANGED ⟨sentEcho, sentReady⟩
37 Next ≜
38     ∃ p ∈ Corr, nextEcho ∈ SUBSET sentEcho, nextReady ∈ SUBSET sentReady :
39       ∧ Receive(p, nextEcho, nextReady)
40       ∧ ∨ SendEcho(p, nextEcho, nextReady)
41         ∨ SendReady(p, nextEcho, nextReady)
42         ∨ Decide(p, nextReady)
43         ∨ UNCHANGED ⟨pc, sentEcho, sentReady⟩
44 NoDecide ≜ ∀ p ∈ Corr : pc[p] ≠ "AC"   Invariant stating that processes never Decide
45 ──────────────────────────────────────────────────────────────
```

Figure 2.1. TLA+ specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [1985].

$rcvdReady = [p \in \{1\} \mapsto \{\}]$, and $sentEcho = sentReady = \{42\}$ is not reachable via any of the transitions defined.

2.3 The Solidity Language

Solidity is the main high-level language specifically designed for smart contracts targeting EVM bytecode. It is a Turing-complete language in which a **`contract`** is a structure similar to a class in object-oriented programming languages. Contracts can have data types such as Boolean, integer, array, and map, and declare both public and private functions, depending on whether they can be called directly by the user. Such functions can make use of common programming languages control structures, such as conditionals and loops, and can be marked as **`payable`**, allowing them to receive funds in ether, Ethereum's native currency, with each contract having its own ether balance.

When deployed, Solidity contracts consist of a storage area and a set of functions. The storage is a persistent memory space used to store variables whose values represent the contract state and functions are the interface by which users interact with the contract. Functions are allowed to access the storage both in read and write modes and their behaviour is defined by their corresponding bytecode instructions, stored persistently in a separate memory residing within the blockchain; the storage management is done via EVM primitives. The interaction with a contract is performed by calling one of its functions, which can call other functions during its execution. The execution costs are commonly paid via a fee in the platform's native currency. Each individual function call is an atomic transaction, i.e., it either executes without raising exceptions, committing its changes, or rolls back completely if an exception occurs, leaving the state unchanged. Contrarily, in traditional programming languages all the changes made by a function prior to throwing an exception are preserved. Further details on Solidity and the operation of its contracts are available in the official language documentation[1].

As an example, consider the `Auction` contract, shown in Figure 2.2, which provides realistic support for an auction. This contract has three state variables, in lines 2-4, `bid` and `cash`, of type unsigned integer, and `winner`, of type **`address`**, which is a 20 byte Ethereum address. To manage the auction, `bid` and `cash` store the current highest bid and the amount of currency gathered, respectively, while `winner` stores the address from which the current highest bid was made. One function is present, `offer`, which handles the placing of new bids by users; additional features, such as the ability to end the auction and forward the gathered

[1] Available at `https://docs.soliditylang.org/en/latest`.

```
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 - 10^15 wei;
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     ...
19 }
```

Figure 2.2. Example of an auction contract in Solidity.

funds to the auctioneer, are abstracted for simplicity. The `offer` function has two implicit arguments: **`msg.value`**, that stores the amount of funds sent to the function, and **`msg.sender`**, that stores the address from which the function was called. Every new offer is subject to a fee of 10^{15} wei[2], in line 7, after which the function checks whether the new bid is greater than the current highest bid, in line 8. A **`require`** statement works as a pre-condition in Solidity, usually employed to filter invalid inputs. In `offer`, if the new bid is not large enough the transaction reverts, with the fee payment being rescinded. After validating the new bid, the function returns the previous highest bid, if available, to its owner, in lines 9-13, and updates the state variables, in lines 14-16. An **`assert`** statement works as a post-condition in Solidity, meaning that its expression should never be false in a valid execution, with a violation leading to a `Panic` exception being thrown. In the example, an assertion error happens if the contract does not have enough funds to return the previous highest bid to its owner. Although both the **`require`** and **`assert`** statements stop the function execution and revert the changes made, they do it via different exception types, with the former doing it gracefully, since

[2]Ethereum's native currency, ether, has wei as its smallest subunit, with 1 US Dollar being worth approximately 238 wei at the time of writing.

a failing `require` is a valid behavior, and the latter resorting to a `Panic` exception, also thrown, for instance, if a division by zero occurs.

2.4 Satisfiability Modulo Theories

SMT studies the decision problem of satisfiability of FOL formulas within the scope of various first-order theories, such as arithmetic, arrays and bit-vectors. It combines the expressiveness of FOL with efficient decision procedures, enabling the representation and analysis of many real world problems. Here the focus is on the aspects of SMT relevant to this dissertation, with further details being available in the comprehensive book by Kroening and Strichman [2016].

Considered formulas are in quantifier-free multi-sorted first-order logic and contain logical operators, predicates, and theory atoms, with examples of theory atoms being equalities over arithmetics and uninterpreted functions. A formula F is treated as a DAG F_{DAG}, where nodes are labelled with symbols, which can be functions, predicates, constants, or logical operations, and outgoing edges are ordered. Formally, a symbol is a tuple $\langle p, s, \mathbf{c} \rangle$ where p is the symbol name, e.g., $+$, $\wedge$, or 0, s the return sort, and $\mathbf{c} = s_1, \ldots, s_n$ are the argument sorts, where n is the arity of p. Hence, if a node is labelled with $\langle p, s, \mathbf{c} \rangle$, it has the ordered outgoing edges to the nodes c_i having the return sorts s_i, for $1 \leq i \leq n$. There is a unique source node, and it must be labelled with a symbol with the Boolean return sort. To avoid exponential blowup, if two subtrees rooted at the nodes r and r' are equal, then $r = r'$.

Given the formula F, an SMT solver attempts to determine whether F_{DAG} is satisfiable or not. The formula F_{DAG} is first converted into an equisatisfiable conjunctive normal form (CNF) formula F_{CNF}, i.e., F_{DAG} is converted into a conjunction of logic clauses over a set of propositional atoms and their negations, while maintaining the first-order theory interpretation of the atoms. The SMT solver then determines the satisfiability of the formula via a search space traversal carried out by a conflict-driven clause-learning (CDCL) SAT solver [Marques-Silva and Sakallah, 1999] operating on F_{CNF}. During the search the SAT solver adds logic clauses to F_{CNF} while maintaining as invariant that if F is satisfiable then F_{CNF} is propositionally satisfiable, i.e., F_{CNF} is satisfiable when the theory interpretations of its atoms are ignored. Therefore, if at some point F_{CNF} becomes unsatisfiable propositionally, F must be unsatisfiable as well.

SMT theories are plentiful, each with their own associated decision procedures that aid the CDCL search space traversal. The operators of theory of arrays [de Moura and Bjørner, 2009] will be used to illustrate the aspects of a

theory. Given the set of sorts S, containing one sort s_τ for each relevant type τ, e.g., integers, an array sort $\mathsf{s}_{\tau_1,\tau_2}$ has the form $\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$, with $\mathsf{s}_{\tau_1} \in \mathsf{S}$ being its index sort and $\mathsf{s}_{\tau_2} \in \mathsf{S}$ being its value sort. Each array sort is supported by two basic operators, *select* : $(\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}, \mathsf{s}_{\tau_1}) \rightarrow \mathsf{s}_{\tau_2}$, which handles array access at a given index, and *store* : $(\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}, \mathsf{s}_{\tau_1}, \mathsf{s}_{\tau_2}) \rightarrow \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$, which updates an array for a given index and value. For brevity, $select(a, i)$ will be written as $a[i]$ in the remainder of the manuscript. Regarding equality between arrays, different interpretations are possible. Arrays with extensionality [Stump et al., 2001] are used in this dissertation, which are considered equal if they contain the same values in the same entries. Extensionality is formally defined as $\forall\ a, b : \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}\ .\ a = b \vee \exists\ i : \mathsf{s}_{\tau_1}\ .\ a[i] \neq b[i]$. For access and update, consistency is ensured by the following property:

$$\forall\ a : \mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2},\ i : \mathsf{s}_{\tau_1},\ j : \mathsf{s}_{\tau_1},\ v : \mathsf{s}_{\tau_2}\ .\ \underbrace{store(a, i, v)[i] = v}_{\text{access consistency}} \wedge \underbrace{(i = j \vee store(a, i, v)[j] = a[j])}_{\text{update consistency}}$$

In addition to *select* and *store*, the theory of arrays can be extended with other operators, two of which are map_f and K_{s_τ}, whose signatures are shown below. The map_f operator applies a n-ary function $f : (\mathsf{s}_{\tau_1}, ..., \mathsf{s}_{\tau_n}) \rightarrow \mathsf{s}_\tau$ to the values stored in each index of its array arguments, producing a new array whose values are the result of the function application, i.e., map_f is the pointwise array extension of f. The K_{s_τ} operator produces a constant array, with all its values being the constant provided as argument. The properties defining the behaviour of these two operators are shown after their signatures.

$$map_f : (\mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_1}, ..., \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_n}) \rightarrow \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_f} \qquad K_{\mathsf{s}_\tau} : \mathsf{s}_{\tau_{const}} \rightarrow \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_{const}}$$

$$\forall\ a_1 : \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_1},\ ...,\ a_n : \mathsf{s}_\tau \Rightarrow \mathsf{s}_{\tau_n},\ i : \mathsf{s}_\tau\ .\ map_f(a_1, ..., a_n)[i] = f(a_1[i], ..., a_n[i])$$

$$\forall\ i : \mathsf{s}_{\tau_1},\ v : \mathsf{s}_{\tau_2}\ .\ K_{\mathsf{s}_{\tau_1}}(v)[i] = v$$

The *select* and *store* operators are part of theory of arrays with extensionality defined in version 2.6 of the SMT-LIB standard [Barrett et al., 2021]. Other operators are provided on a solver-by-solver basis, e.g., Z3 [de Moura and Bjørner, 2008b] supports both map_f and K_{s_τ}, while CVC5 [Barbosa et al., 2022a] supports K_{s_τ}; future SMT-LIB updates may add these operators to the standard.

2.5 Constrained Horn Clauses

The constrained Horn clauses formalism is a logic-based language suited for verification tasks such as safety, termination, and loop invariant computation. Here a CHC characterization based on FOL and the fixed-point operator adapted from the work of Blass and Gurevich [1987] is presented.

To precisely define how CHC are used we must first define some notation. Let ψ be a first-order formula over a theory T, with free variables $\boldsymbol{x} = \{x_1, ..., x_n\}$, and $\{\mathcal{P}_1, \ldots, \mathcal{P}_m\}$ be a finite set of predicates over $\boldsymbol{x}$, such that no $\mathcal{P}_i$ appears in ψ. A predicate $\mathcal{P}(\boldsymbol{x})$ over a set of variables $\boldsymbol{x}$ is associated with a so called interpretation, that states on which values of $\boldsymbol{x}$ the predicate is true. The interpretation can be thought of as a set of tuples of length $|\boldsymbol{x}|$ explicitly stating such values. The satisfiability of $\mathcal{P}_1(\boldsymbol{x}) \wedge \ldots \wedge \mathcal{P}_m(\boldsymbol{x}) \wedge \psi(\boldsymbol{x})$ in theory T, with the interpretations of $\mathcal{P}_i$ being $\Delta_{\mathcal{P}_i}$, is denoted by $\bigcup_{i=1}^{m} \Delta_{\mathcal{P}_i} \models_T \mathcal{P}_1(\boldsymbol{x}) \wedge \ldots \wedge \mathcal{P}_m(\boldsymbol{x}) \wedge \psi(\boldsymbol{x})$.

When modelling a program, the predicates $\mathcal{P}_i$ are chosen to represent reachable states in certain key positions. These include the program counter values corresponding to the starts and exits of loops, function call sites, and, depending on the chosen modelling approach, starts and ends of conditional branches. Complementary, the first-order formulas ψ encode the effect that the program code executed between the positions represented by predicates has on the state. The interpretations of the predicates are, however, not explicitly known, being defined implicitly by the program code represented by the formulas ψ and how the predicates are related by these formulas. CHC provide a way of representing the relations between the program code and predicates, and there are highly engineered implementations of algorithms for determining over-approximations of the interpretations of the predicates for a given CHC system.

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

$$\mathcal{H}(\boldsymbol{x}) \leftarrow \exists \boldsymbol{y}. \mathcal{P}_1(\boldsymbol{y}) \wedge \ldots \wedge \mathcal{P}_m(\boldsymbol{y}) \wedge \phi(\boldsymbol{x}, \boldsymbol{y}), \text{ for } m \geq 0 \qquad \text{(DefClause)}$$

where ϕ is a first-order formula over $\boldsymbol{x}, \boldsymbol{y} \subseteq \mathcal{V}$, with respect to the theory T, $\boldsymbol{x}$ are the variables free in ϕ, $\mathcal{H} \in \mathscr{P}$ is a predicate with arity matching $\boldsymbol{x}$, $\mathcal{P}_i \in \mathscr{P}$ are predicates with arities matching $\boldsymbol{y}$, and no predicate in $\mathscr{P}$ appears in ϕ. For a clause c we write $\mathit{head}(c) = \mathcal{H}$ and $\mathit{body}(c) = \exists \boldsymbol{y}. \mathcal{P}_1(\boldsymbol{y}) \wedge \ldots \wedge \mathcal{P}_m(\boldsymbol{y}) \wedge \phi(\boldsymbol{x}, \boldsymbol{y})$.

As an example, consider the program statement `x = x + 1`. To represent the states two copies of the program variable `x` are needed, one representing `x`'s value before the execution of the program statement and another representing the value after the execution. By convention these are represented by first-order

variables x and x', respectively. The predicates $\mathcal{P}$ and $\mathcal{Q}$, that hold before and after the execution of the increment, are then defined. The CHC system modelling this fragment would then be $\mathcal{Q}(x') \leftarrow \exists x.\mathcal{P}(x) \wedge x' = x + 1$.

To understand how predicate interpretations are obtained, consider the following. For each $\mathcal{P} \in \mathscr{P}$ we define the transfinite sequence $\Delta^{\alpha}_{\mathcal{P}}$ given by

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

Since $\Delta^{\alpha}_{\mathcal{P}}$ is monotonic, there is a value for α such that $\Delta^{\alpha}_{\mathcal{P}} = \Delta^{\alpha+1}_{\mathcal{P}}$. The set $\Delta^{\alpha}_{\mathcal{P}}$ with this property is denoted by $\Delta_{\mathcal{P}}$.

When checking safety, we are interested in determining whether a bad state is reachable. The bad state is represented by a CHC with a special head symbol $\bot$ and a body describing the bad state in logic. Determining whether the bad state is reachable reduces then to determining whether the interpretation $\Delta_{\bot}$ of predicate $\bot \in \mathscr{P}$ is empty. To continue the above example, we might be interested whether after executing the increment, the value of x can be greater than 255. This would be encoded as the CHC $\bot \leftarrow \exists x.\mathcal{Q}(x) \wedge x > 255$.

Modern CHC solvers, based on the IC3 verification algorithm [Bradley, 2011], guarantee that if $\Delta_{\bot}$ is nonempty then the model of a program violates a safety property, and we are able to map predicate interpretations to a program execution. Conversely, if $\Delta_{\bot}$ is empty, either the solver does not terminate, or it provides quantifier-free first-order formulas $\eta_{\mathcal{P}}(\boldsymbol{x})$ in T for each $\mathcal{P} \in \mathscr{P}$ that serve as safe inductive invariants in the following sense:

1. Each formula $\eta_{\mathcal{P}}$ over-approximates an interpretation $\Delta_{\mathcal{P}}$, meaning that $\Delta_{\mathcal{P}} \models_T \mathcal{P}(\boldsymbol{x}) \implies \eta_{\mathcal{P}}(\boldsymbol{x})$.

2. For each clause $c \in S$ of the form DefClause,

   (a) if $head(c) \neq \bot$, then $\models_T \eta_{\mathcal{P}_1}(\boldsymbol{y}) \wedge \ldots \wedge \eta_{\mathcal{P}_m}(\boldsymbol{y}) \wedge \phi(\boldsymbol{x},\boldsymbol{y}) \implies \eta_{\mathcal{H}}(\boldsymbol{x})$;

   (b) if $head(c) = \bot$, then $\models_T \neg\big(\eta_{\mathcal{P}_1}(\boldsymbol{y}) \wedge \ldots \wedge \eta_{\mathcal{P}_m}(\boldsymbol{y}) \wedge \phi(\boldsymbol{x},\boldsymbol{y})\big)$.

Following the terminology set by Bjørner et al. [2015], a set of CHC is called satisfiable if $\Delta_{\bot}$ is empty, and unsatisfiable otherwise.

When presenting CHC systems and individual clauses in this manuscript some conventions are used 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 do not appear in the formulas are not written.

Third, superfluous equalities are omitted, e.g., if an element y_i of $\boldsymbol{y}$ is equated with an element x_j of $\boldsymbol{x}$ in a top-level conjunct of ϕ, then the equality is not written and instead y_i is substituted for x_j in the head.

To summarise, CHC allow for a natural modelling of programs. Each predicate $\mathcal{P}$ describes the set of reachable states in the points of interest in the program, and correspond to some concrete program counter values. The CHC encode the flow of control between these points, and their constraints ϕ encode the conditions for control flow and the effects of the executions. The safety properties can be encoded using the special predicate $\bot$ that by convention can only appear as a head of CHC together with bodies that represent the negations of the safety properties. Finally, the operator Δ is used for accumulating the reachable states in the predicates appearing as CHC heads.

Chapter 3

Symbolic Model Checking for TLA$^+$ Made Faster

The ability to ensure that a distributed system, such as a blockchain, operates correctly is paramount. To achieve this, one can specify and reason about protocols using TLA$^+$ [Lamport, 2002], with this approach being adopted by companies such as Amazon Web Services [Newcombe et al., 2015]. Despite interest and recent advances, the verification of distributed systems remains notoriously difficult. This is due to the fact that, given their distributed nature, distributed algorithms' executions admit numerous potential interleavings of steps, with state-spaces generally growing exponentially with the number of participants.

A handful of tools are available to aid in verifying TLA$^+$ specifications [Konnov et al., 2022]. TLC [Yu et al., 1999] is an explicit state model checker that enumerates all reachable states of the given system. APALACHE [Konnov et al., 2019] is a symbolic bounded model checker that uses a SMT encoding of states in order to better tackle the state explosion problem. TLAPS [Chaudhuri et al., 2010] is an interactive proof system that enables the proving of properties without the need of exploring the state-space itself. Despite providing the benefit of verifying specifications with infinite state-spaces, and efforts being made towards partial automation [Merz and Vanzetto, 2018], TLAPS adoption is still slow, with engineers favouring the push-button automation provided by model checkers.

This chapter focuses on symbolic model checking for TLA$^+$, as spearheaded by the SMT encoding which underpins APALACHE, but provide insights into SMT-based model checking that may generalise to other contexts. The encoding of TLA$^+$ into SMT done by APALACHE removes all structural information present in the encoded specification, with the TLA$^+$ data structures being represented via uninterpreted constants in the generated SMT formula. The information not

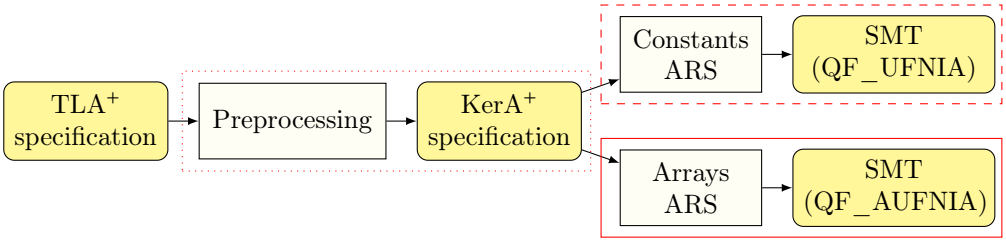


Figure 3.1. Overview of the symbolic model checking approach for TLA$^+$. The dotted box highlights the identification of symbolic transitions proposed by Kukovec et al. [2018] and the rewriting into KerA$^+$. The dashed box highlights the encoding based on uninterpreted constants by Konnov et al. [2019]. The solid box highlights the arrays-based encoding propose in this work.

forwarded to the SMT solver has the potential to significantly improve solving efficiency. This work proposes an alternative encoding that makes full use of the SMT theory of arrays [de Moura and Bjørner, 2009] to encoded the main TLA$^+$ data structures, i.e., sets and functions, with the goal of improving solving performance, which is the determining factor in overall model checking performance.

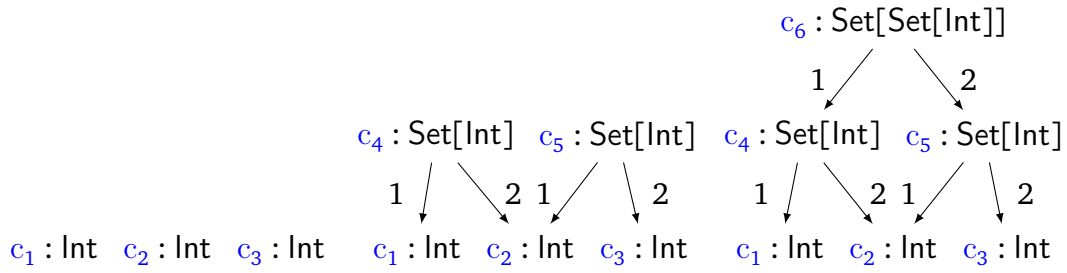
3.1 Symbolic Model Checking Approach

The novel encoding consists of a new abstract reduction system (ARS) to generate constraints in the SMT theory of arrays. It relies on APALACHE's preprocessing infrastructure, which rewrites the input specification into the KerA$^+$ verification-friendly fragment of TLA$^+$ [Konnov et al., 2019] and then applies ARS rules to generate the SMT formula to be solved, as illustrated in Figure 3.1.

3.1.1 The KerA$^+$ Language

TLA$^+$ provides users with a myriad of ways of specifying systems. This richness, although being one of its strengths, adds significant difficulty to the generation of SMT constraints. To overcome this challenge, TLA$^+$ specifications are rewritten into a more compact language, KerA$^+$, before being checked. From KerA$^+$, the ARS can generate SMT constraints in a simpler and provably sound way.

The KerA$^+$ language consists of a small subset of TLA$^+$ conjoined with four additional constructs not originating from TLA$^+$, and is able to express almost all TLA$^+$ expressions. It contains constructs for the manipulation of sets, functions, records, tuples, and sequences, as well as integer arithmetic operators, Boolean and integer literals, and constants, with all data structures having a bounded size.



(a) integers 5, 6, and 7 **(b) sets of integers** $\{5,6\}$ and $\{6,7\}$ **(c) set of sets of integers** $\{\{5,6\},\{6,7\}\}$

Figure 3.2. Illustration of three arenas. The modelled elements have the overapproximation $c_1 = 5$, $c_2 = 6$, $c_3 = 7$, $c_4 = \{5,6\}$, $c_5 = \{6,7\}$, and $c_6 = \{\{5,6\},\{6,7\}\}$. Note that the concrete value of a cell can be given by any of the possible subtrees having said cell as a root, e.g., for c_6 we have that $\exists c_4 \in \mathcal{P}(\{5,6\}), c_5 \in \mathcal{P}(\{6,7\})\ .\ c_6 \in \mathcal{P}(\{c_4, c_5\})$; here $\mathcal{P}$ stands for power set.

The semantics of KerA$^+$ derive directly from the TLA$^+$ constructs it uses, with the non-TLA$^+$ based constructs, which help simplify the rewriting system, having simple control semantics. The correctness of the rewriting itself is guaranteed by construction. One example is the rewriting of $S \cup T$ into the set comprehension $\{x \in S : x \in T\}$. Further KerA$^+$ details are available in Appendix A.

3.1.2 Abstract Reduction System

In order to verify a specification in KerA$^+$ a SMT formula that is equisatisfiable to it is generated. To do so, an abstract reduction system is used, which iteratively applies reduction rules that transform KerA$^+$ expressions into SMT constraints. The core of the ARS is the *arena*, a graph structure that overapproxiamtes the specification's data structures and guides rule application. The rules collapse KerA$^+$ expressions into *cells*, which represent the symbolic evaluation of these expressions, with the cells then being used as vertices in the arena. The arena edges represent the data structures overapproximation, e.g., a cell representing a set will have directed edges to the cells representing all its potential elements, as illustrated in Figure 3.2. The reduction process terminates when the initial KerA$^+$ expression e is collapsed into a single cell c, producing a SMT formula Φ in the process, such that $c \wedge \Phi$ is equisatisfiable to e; equisatisfiability relies on the boundedness of the data structures and is detailed in Section 3.2.3. The satisfiability of e can then be checked by forwarding $c \wedge \Phi$ to a SMT solver.

Formally, the ARS is defined as $(\mathcal{S}, \rightsquigarrow)$, with $\mathcal{S}$ being the set of ARS states and

$\rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S}$ being the transition relation. A state $(e, \mathcal{A}, \nu, \Phi) \in \mathcal{S}$ is a four-tuple containing a KerA$^+$expression e, an arena $\mathcal{A}$, a binding of names to cells ν, and a first-order formula Φ. ARS states' elements contain a number of cells, which are first-order terms annotated with a type τ. Cells of type Bool and Int are interpreted in SMT as Booleans and integers, while cells of the remaining types are encoded as uninterpreted constants in the constants encoding; the arrays encoding approach is discussed in Section 3.2. Cells are referred to via the notation c_{name} or c_{index}, and they can be seen as both KerA$^+$ constants and first-order terms in SMT. An arena is a directed acyclic graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ being a finite set of cells and $\mathcal{E} \subseteq \mathcal{V} \times (1..|\mathcal{V}|) \times \mathcal{V}$ being a set of relations between the cells in $\mathcal{V}$. Every relation between cells is represented by an arena edge of form $(\mathrm{c}_a, i, \mathrm{c}_b)$, also written $\mathrm{c}_a \xrightarrow{i} \mathrm{c}_b$, with no duplicates, i.e., for every pair $(\mathrm{c}_{a_1}, i_1, \mathrm{c}_{b_1}), (\mathrm{c}_{a_2}, i_2, \mathrm{c}_{b_2}) \in \mathcal{E}$ we have that $\mathrm{c}_{a_1} = \mathrm{c}_{a_2} \wedge \mathrm{c}_{b_1} \neq \mathrm{c}_{b_2}$ implies $i_1 \neq i_2$, and no gaps in the relation indexes, i.e., for every edge $(\mathrm{c}_a, i, \mathrm{c}_b)$ and index $j \in 1..(i-1)$ we have that $\exists \mathrm{c}_c \in \mathcal{V} \,.\, (\mathrm{c}_a, j, \mathrm{c}_c)$. A binding is a partial function from KerA$^+$ variables to $\mathcal{V}$ of $\mathcal{A}$, i.e., a mapping from variables to cells. Finally, Φ is a formula in the SMT fragment supported by the ARS and the target SMT solver, e.g., the quantifier-free uninterpreted functions and nonlinear arithmetics (QF_UFNIA) fragment supported by the constants encoding.

A series of n reduction steps has the form $s_0 \rightsquigarrow \ldots \rightsquigarrow s_n$, with each step generating state s_{i+1} for state s_i, $0 \leq i < n$, by applying a reduction rule. The initial state $s_0 = (e_0, \mathcal{A}_0, \nu_0, \Phi_0)$ has e_0 as the initial KerA$^+$ specification, $\mathcal{A}_0 = (\emptyset, \emptyset)$, ν_0 containing no mappings, and $\Phi_0 = \mathit{true}$. The reduction steps end upon reaching a state $s_n = (e_n, \mathcal{A}_n, \nu_n, \Phi_n)$, with e_n being a single cell $\mathrm{c} \in \mathcal{V}_n$ and $\mathcal{A}_n = (\mathcal{V}_n, \mathcal{E}_n)$. Below three examples of rules are given.

Integer literal reduction. One of the simplest rules has an integer literal *num* being rewritten into a cell c_{num}. This cell is added to the arena and a constraint equating c_{num} to the literal is conjoined with Φ; we use vertical lines to separate state elements and commas to indicate additions to $\mathcal{A}$ and conjunctions to Φ.

$$\frac{\left\langle num : \mathsf{Int} \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \qquad num \text{ is one of } 0, 1, -1, \ldots}{\left\langle \mathrm{c}_{num} \mid \mathcal{A}, \mathrm{c}_{num} : \mathsf{Int} \mid \nu \mid \Phi, \mathrm{c}_{num} = num \right\rangle} \text{ (INT)}$$

The descriptions of rules can be given as inferences, with the premisses above the bar and the resulting state below it. Inferences, although reasonable to express rules such as INT, are not suitable to give the intuition about how more complex rules work. In light of this, a simplified notation will be used moving forward. Inferences are inlined as $\rightarrowtail$ and nonessential information is omitted,

e.g., propagated values. Below we can see rule INT in this simplified format. Note that only $\mathcal{A}$ and Φ updates are shown, without propagating them, and that ν is omitted. All rules are available as inferences in Appendix B.

$$\frac{num : \mathsf{Int} \quad num \text{ is one of } 0, 1, -1, \ldots}{\rightarrowtail \mathrm{c}_{num} \mid \mathrm{c}_{num} : \mathsf{Int} \mid \mathrm{c}_{num} = num} \quad \texttt{(Int)}$$

Picking. To pick a cell out of n cells an oracle θ is used, as per rule `FromBasic`. In addition to the FROM ... BY θ expression, this rule requires that all pickable cells are of the same basic type τ, e.g., Int. The resulting state has a new cell c_{pick}, which is equated to one of the n cells if $1 \leq \theta \leq n$ and is unconstrained otherwise. Picking among cells representing data structures, e.g., sets, can be done via a more general version of rule `FromBasic`.

$$\frac{\mathsf{FROM}\ \mathrm{c}_1, \ldots, \mathrm{c}_n\ \mathsf{BY}\ \theta : \tau \quad \tau \text{ is basic and } \mathrm{c}_1 : \tau, \ldots, \mathrm{c}_n : \tau}{\rightarrowtail \mathrm{c}_{pick} \mid \mathrm{c}_{pick} : \tau \mid \bigwedge_{1 \leq i \leq n} (\theta = i \rightarrow \mathrm{c}_{pick} = \mathrm{c}_i)} \quad \texttt{(FromBasic)}$$

Branching. To reduce an if-then-else expression we rely on picking, as can be seen in rule `ITE`; parentheses indicate the application of another rule. This rule has an inference that picks one of the branches via an oracle, with the branching itself being made by setting the oracle value via the constraint $\theta = 1 \leftrightarrow \mathrm{c}_p$.

$$\begin{aligned} \mathsf{ITE}(\mathrm{c}_p, \mathrm{c}_1, \mathrm{c}_2) : \tau \rightarrowtail & \left(\mathsf{FROM}\ \mathrm{c}_1, \mathrm{c}_2\ \mathsf{BY}\ \theta : \tau \mid \theta : \mathsf{Int} \mid 1 \leq \theta \leq 2 \rightarrowtail \mathrm{c}_{res} \right) \\ & \rightarrowtail \mathrm{c}_{res} \mid \theta = 1 \leftrightarrow \mathrm{c}_p \end{aligned} \quad \texttt{(ITE)}$$

3.2 Encoding TLA$^+$ using Arrays

The goal is to encode TLA$^+$ data structures in a structure-preserving way. To do this, arrays are used to represent the main components of TLA$^+$, sets and functions, as SMT constraints. We follow the ARS structure described in Section 3.1.2, but update the reduction rules handling sets and functions. The remaining TLA$^+$ constructs, e.g., tuples, are represented as per the constants encoding; details on the SMT constraints generated by constants encoding are in Appendix C.

The two efficiency benefits of the arrays encoding are the ease of access of data structures and the possibility of using SMT equality. The first benefit can be easily understood by the use of SMT *select*, which allows us to check a stored

value by using a single constraint, in contrast to the amount of constraints used in the constants encoding, which is linear in the size of data structures' overapproximation. The second benefit affects the comparison of data structures, which can be done via a single SMT equality for sets and functions in the arrays encoding, since these structures are represented by a single SMT term, while the constants encoding requires a number of constraints that is quadratic in the size of data structures' overapproximation. A comparison between the two encodings is shown in Table 3.1. We first describe how to encode sets and functions, and then present the correctness argument for the reduction to arrays.

Table 3.1. Amount of constraints generated by each SMT encoding to model the main TLA$^+$ constructs.

Construct	**Constants**	**Arrays**
Set enumeration	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set filter	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set map	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set membership	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Set equality	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Fun. definition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Fun. domain	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Fun. equality	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Fun. update	$\mathcal{O}(n)$	$\mathcal{O}(1)$
Fun. application	$\mathcal{O}(n)$	$\mathcal{O}(n)$

3.2.1 Encoding TLA$^+$ Sets using Arrays

Arrays are used to encode TLA$^+$ sets as characteristic functions, i.e., a set of type τ is represented by an array of sort $\mathsf{s}_\tau \Rightarrow$ Bool. Set membership is encoded by storing *true* or *false* on a given array index. The reduction rules used to handle the main set operators are presented below.

Set Enumeration. The simplest way to create a set is to enumerate its elements. Rule `Enum` reduces an explicit set of cells to a fresh cell c_{set}, whose edges link it to its elements; $\mathrm{c}_{set} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n$ is a shorthand for $\mathrm{c}_{set} \xrightarrow{1} \mathrm{c}_1, \ldots, \mathrm{c}_{set} \xrightarrow{n} \mathrm{c}_n$. There is no guarantee that the enumerated elements are unique, thus the arena may contain edges to repeated elements.

$$\{\mathrm{c}_1, \ldots, \mathrm{c}_n\} : \mathsf{Set}[\tau] \rightarrowtail \mathrm{c}_{set} \mid \mathrm{c}_{set} : \mathsf{Set}[\tau], \mathrm{c}_{set} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n \mid \mathit{EnumCtr} \qquad (\mathtt{Enum})$$

The constraints *EnumCtr* added by the arrays encoding create an empty set, by using a constant array with the value *false*, $\bot$, and updates the array by storing *true*, $\top$, on the appropriate indexes. The array resulting from the last update, $a^n_{\mathrm{c}_{set}}$, is then equated to c_{set}. Since cells representing repeated elements lead to updates to the same index, we encode standard sets, in contrast the constants encoding,

which encodes multisets due to the arena imprecision; multisets lead to multiple constraints being generated to encode membership of a single element.

$$\underbrace{a^0_{\mathrm{c}_{set}} = K_\tau(\bot)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1\le i\le n} a^i_{\mathrm{c}_{set}} = store(a^{i-1}_{\mathrm{c}_{set}}, \mathrm{c}_i, \top)}_{\text{set updates}} \wedge \underbrace{\mathrm{c}_{set} = a^n_{\mathrm{c}_{set}}}_{\text{cell equality}} \qquad (EnumCtr)$$

Although the amount of constraints generated by the arrays encoding to model set enumeration is equal to that of the constants encoding, it has the benefit of generating a defined interpretation for c_{set}, the array $a^n_{\mathrm{c}_{set}}$, which is not present in the constants encoding. This has a significant impact on set membership and cell equality, as described below.

Set Membership. The checking of a membership relation $\mathrm{c}_x \in \mathrm{c}_{set}$, given the presence of the arena edges $\mathrm{c}_{set}\rightarrow\mathrm{c}_1, ..., \mathrm{c}_n$ and $1 \le x \le n$, is straightforward. A single fresh cell of Boolean type is introduced and is equated to $\mathrm{c}_{set}[\mathrm{c}_x]$.

Cell Equality. The constraints generated by encoding set membership and many other constructs assume that cells can be compared. When this is not directly the case the equalities are cached in preparation. For example, if a set of n tuples c_t of size two is being equated, the constraints $\mathrm{c}_{t_i} = \mathrm{c}_{t_j} \leftrightarrow \mathrm{c}^1_{t_i} = \mathrm{c}^1_{t_j} \wedge \mathrm{c}^2_{t_i} = \mathrm{c}^2_{t_j}$, with $1 \le i \le n$ and $1 \le j \le n$, are added to Φ; here we use c^1_t and c^2_t to represent the values of the 2-tuple. The need for this caching of equalities only arises when data structures encoded as uninterpreted constants are compared. For the remaining rules it is assumed that caching was done, if needed, and cells can be compared via direct equality.

Set Filter. In TLA$^+$, the elements of a set S can be filtered by a predicate p via the expression $\{x \in S : p\}$. This expression will create a set F which contains only the elements of S that satisfy p, e.g., $\{x \in \{-1, 0, 1\} : x \ge 0\} = \{0, 1\}$. Rule `Filter` reduces a filter to a new set cell, c_F, whose arena overappoximation contains the elements of S, but whose constraints ensure that only filtered elements are members of F; $p[y/x]$ means that x is replaced by y in p and parentheses indicate the application of another rule, the predicate resolution rule in this case.

$$\begin{aligned}\{x \in \mathrm{c}_S : p\} : \mathsf{Set}[\tau] \text{ and } \mathrm{c}_S\rightarrow\mathrm{c}_1, \ldots, \mathrm{c}_n \\ \rightarrowtail \big(\ p[\mathrm{c}_1/x] : \mathsf{Bool}, \ldots, p[\mathrm{c}_n/x] : \mathsf{Bool} \rightarrowtail \mathrm{c}^p_1, \ldots, \mathrm{c}^p_n\ \big) \\ \rightarrowtail \mathrm{c}_F \mid \mathrm{c}_F : \mathsf{Set}[\tau], \mathrm{c}_F\rightarrow\mathrm{c}_1, \ldots, \mathrm{c}_n \mid FilterCtr\end{aligned} \qquad (\texttt{Filter})$$

The constraints added use an array $a^0_{c_F}$ initially unconstrained, i.e., the values mapped by all the indexes of $a^0_{c_F}$ are unconstrained, as opposed to $a^0_{c_{set}}$ in *EnumCtr*. The values of $a^0_{c_F}$ mapped by indexes $c_1, \ldots, c_n$ are constrained by $c^p_1, \ldots, c^p_n$ via array access, i.e., $a^0_{c_F}[c_i]$ is asserted to be *true* or *false* based on c^p_i, with $1 \leq i \leq n$. Pointwise conjunction is then applied to c_S and $a^0_{c_F}$ via the map_f SMT operator; we go from a^0_F to a^n_F to keep the array index in step with the arena overapproximation. Indexes whose values were *false* in S remain so in F, and indexes whose values were *true* in S store the filter's predicate evaluation.

$$\underbrace{\bigwedge_{1 \leq i \leq n} \text{ite}\left(c^p_i, a^0_{c_F}[c_i], \neg a^0_{c_F}[c_i] \right)}_{\text{predicate-based constraining}} \wedge \underbrace{a^n_{c_F} = map_\wedge(c_S, a^0_{c_F})}_{\text{pointwise conjunction}} \wedge \underbrace{c_F = a^n_{c_F}}_{\text{cell equality}} \qquad (\textit{FilterCtr})$$

Both encodings generate a linear amount of constraints, since n $p[c_i/x]$ predicates have to be considered. Unlike with *EnumCtr*, *FilterCtr* does not contain many *store* operations, due to the usage of map_f. This avoids the need to create intermediary arrays, and is not possible in *EnumCtr* due to its constant array.

Set Map. The expression $\{e : x \in S\}$ can be used to construct a set M from a set S, having all the elements of M as $e[y/x]$, with $y \in S$. For example, the expression $\{x \div 5 : x \in \{4, 5, 6\}\}$ yields the set $\{0, 1\}$, with $\div$ denoting standard integer division. To reduce set map rule `Map` is used.

$$\begin{aligned} \{e : x \in c_S\} : \mathsf{Set}[\tau] \text{ and } c_S \rightarrowtail c_1, \ldots, c_n \\ \rightarrowtail \left(e[c_1/x] : \tau, \ldots, e[c_n/x] : \tau \rightarrowtail c^e_1 :, \ldots, c^e_n \right) \\ \rightarrowtail c_M \mid c_M : \mathsf{Set}[\tau], c_M \rightarrowtail c^e_1, \ldots, c^e_n \mid MapCtr \end{aligned} \qquad (\texttt{Map})$$

The constraints added in rule `Map` are similar to those added in rule `Enum`. The difference between them is that set enumeration precisely defines the elements to be added to the new set cell, while set map is based on an existing set cell, which is a set overapproximation. Due to this, membership in M has to be guarded by membership in S, leading to a linear amount of constraints being generated.

$$\underbrace{a^0_{c_M} = K_\tau(\bot)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1 \leq i \leq n} \text{ite}\left(\begin{array}{c} c_S[c_i], \\ a^i_{c_M} = store(a^{i-1}_{c_M}, c^e_i, \top), \\ a^i_{c_M} = a^{i-1}_{c_M} \end{array} \right)}_{\text{set updates}} \wedge \underbrace{c_M = a^n_{c_M}}_{\text{cell equality}} \qquad (\textit{MapCtr})$$

3.2.2 Encoding TLA$^+$ Functions using Arrays

Arrays are used to encode TLA$^+$ functions directly as functions themselves. To do this, arrays are used in their general format, with a function $f : \mathsf{s}_{\tau_1} \to \mathsf{s}_{\tau_2}$ being encoded as an array of sort $\mathsf{s}_{\tau_1} \Rightarrow \mathsf{s}_{\tau_2}$. Since functions with a finite domain can rely on infinite sorts, e.g., the integer numbers, the encoding of each function also includes constraints defining its domain set, by means of the rules described in the previous section; the result of a function application to a value outside its domain is undefined in TLA$^+$. This approach allows us to generate SMT constraints that follow directly from TLA$^+$, making the encoding not only more efficient, but also more natural to describe. In contrast, the constants encoding represents functions explicitly as sets of pairs of form $\{\langle x, f(x)\rangle : x \in \text{DOMAIN} f\}$. Due to this, its function manipulation relies on set manipulation, e.g., function comparison is encoded as set comparison, leading to a quadratic amount of constraints. The reduction rules used to handle functions are presented below.

Function Definition. The definition of a function in TLA$^+$ is an expression of the form $[x \in S \mapsto e]$, which maps every domain value v to the expression $e[v/x]$. This definition is similar to that of set map $\{e : x \in S\}$, and thus generates constraints in a similar fashion to rule `Map`. The main difference is that the evaluations of the expression $e[v/x]$ are stored as array values, rather than array indexes, i.e., function definition uses $store(a, v, e[v/x])$ and set map uses $store(a, e[v/x], \top)$, with v being a value in the function's domain or the set being mapped. Every encoded function has a single argument, with multiple arguments being rewritten as tuples in preprocessing.

Unlike with set cells, a function cell c_F in the arena does not directly point to its values, with the arrays encoding adding two edges to c_F, $\mathrm{c}_F \xrightarrow{1} \mathrm{c}_{F_{dom}}$ and $\mathrm{c}_F \xrightarrow{2} \mathrm{c}_{F_{pairs}}$. Cell $\mathrm{c}_{F_{dom}}$ represents the function's domain and cell $\mathrm{c}_{F_{pairs}}$ represents the set of pairs $\{\langle x, f(x)\rangle : x \in \text{DOMAIN} f\}$. Cell $\mathrm{c}_{F_{pairs}}$, despite being in the arena, has no SMT constraints modelling it in the arrays encoding, with its sole purpose being to help propagate the arena edges of the function's codomain elements.

Function Domain. Accessing a function's domain is trivial in the arrays encoding, since the domain set is generated during function definition. This results in a simple access to the array representing the domain.

Function Update. The update of a TLA$^+$ function f is done by changing the result of applying f to an argument arg, $f[arg]$, to be a given value v, via the expression $[f \text{ EXCEPT! } [arg] = v]$. The update will produce a new function g

which is identical f, except that $g[arg] = v$ if $arg \in \text{DOMAIN } f$. The arrays encoding generates a single array update constraint in this case.

Function Application. The application of a function to an argument arg is conceptually simple, but is quite intricate to realize, as can be seen in rule FunApp. The arrays encoding uses an oracle to check that c_{arg} is in the domain and to gather the arena edges of c_{res}. The *FunAppCtr* constraints ensure that the oracle chooses the correct index and equates the result cell to an array access on c_F. Note that the value of c_{res} comes directly from the function application expression itself, with the oracle only been needed to gather the arena edges of c_{res}, if $m > 0$, via c^p. The need for an oracle is restricted to functions whose codomain contain structured data, e.g., $f : \mathsf{Int} \to \mathsf{Set}[\mathsf{Int}]$. If this is not the case, e.g., $g : \mathsf{Int} \to \mathsf{Int}$, rule FunApp is simplified and *FunAppCtr* becomes $\mathrm{c}_{res} = \mathrm{c}_F[\mathrm{c}_{arg}]$.

$$\begin{aligned} \mathrm{c}_F[\mathrm{c}_{arg}] : \tau \text{ and } \mathrm{c}_F \xrightarrow{1} \mathrm{c}_{F_{dom}} \to \mathrm{c}^d_1, \ldots, \mathrm{c}^d_n \text{ and } \mathrm{c}_F \xrightarrow{2} \mathrm{c}_{F_{pairs}} \to \mathrm{c}^p_1, \ldots, \mathrm{c}^p_n \\ \rightarrowtail \begin{array}{l} \big(\ \mathsf{FROM}\ \mathrm{c}^p_1, \ldots, \mathrm{c}^p_n\ \mathsf{BY}\ \theta : \langle \tau_{arg}, \tau \rangle \mid \theta : \mathsf{Int} \mid 0 \le \theta \le n \rightarrowtail \mathrm{c}^p \ \big) \\ \quad \text{and } \mathrm{c}^p[2] \to \mathrm{c}_1, \ldots, \mathrm{c}_m \end{array} \\ \rightarrowtail \mathrm{c}_{res} \mid \mathrm{c}_{res} : \tau, \mathrm{c}_{res} \to \mathrm{c}_1, \ldots, \mathrm{c}_m \mid \mathit{FunAppCtr} \end{aligned} \tag{FunApp}$$

$$\underbrace{\bigwedge_{1 \le i \le n} \wedge \begin{array}{l} (\theta = i \to \mathrm{c}_{arg} = \mathrm{c}^d_i \wedge \mathrm{c}_{F_{dom}}[\mathrm{c}^d_i]) \\ (\theta = 0 \to \mathrm{c}_{arg} \neq \mathrm{c}^d_i \vee \neg \mathrm{c}_{F_{dom}}[\mathrm{c}^d_i]) \end{array}}_{\text{oracle constraining}} \underbrace{\wedge\, \mathrm{c}_{res} = \mathrm{c}_F[\mathrm{c}_{arg}]}_{\text{cell equality}} \qquad (\mathit{FunAppCtr})$$

3.2.3 Correctness of the Reduction to Arrays

Correctness of the ARS is given by four properties: finiteness of the models, compliance to the target SMT theories, termination of any reduction sequence, and soundness of the reductions. These properties had their correctness sketched for the constants encoding by Konnov et al. [2019], with detailed proofs being written by Tran [2023]. Since we rely on the existing ARS and restrict our changes to mainly affect constraint generation, we have the same degree of overapproximation and the correctness arguments made for the constants encoding are in large part valid for the arrays encoding. We present below the definition of a KerA$^+$ model and detail, for each property, how the use of arrays affects the correctness arguments and how they can be adjusted to remain valid.

Models. Every satisfiable KerA$^+$ formula has a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the model domain, consisting of a disjoint union of sets $\mathcal{D}_1, ..., \mathcal{D}_n$, with $\mathcal{D}_i$, $1 \leq i \leq n$, containing the values for type τ_i, and $\mathcal{I}$ is the model interpretation, consisting of assignments of domain values to KerA$^+$ constants. Models are used to access cell values, with the value of a KerA$^+$ expression e in model $\mathcal{M}$ being $[\![e]\!]^{\mathcal{M}}$. In $s_{before} \rightsquigarrow s_{after}$, we go from $\mathcal{M}_{before}$ to $\mathcal{M}_{after}$, with $\mathcal{M}_{after}$ containing the interpretation of additional constants and being thus an extension of $\mathcal{M}_{before}$.

Finiteness. This property states that every interpretation of a KerA$^+$ expression is defined only over finite values. Its proof is derived from the finiteness of the elements being modelled. In the arrays encoding, arrays with infinite sorts are potentially used, e.g., the integers, but all SMT interpretations that can be derived from such arrays are finite, since only finite TLA$^+$ data structures are encoded. This guarantees finiteness of all KerA$^+$ models in the arrays encoding.

Theory Compliance. This property states that any sequence of states $s_0 \rightsquigarrow ... \rightsquigarrow s_n$ has the formulas Φ_i, $1 \leq i \leq n$, in the first-order logic fragment containing only quantifier-free expressions over uninterpreted functions and integer arithmetic. Its proof is done by induction on the constraints generated. The constraint Φ_0 is always *true* and is thus trivially compliant. The inductive case is proved by showing that the constraints added by each rule are compliant. The rules in the arrays encoding only add array constraints, in addition to constraints supported by the constants encoding, so theory compliance is straightforward to guarantee.

Termination. This property states that every sequence of ARS reductions is finite, i.e., the reduction process always terminates. Its proof is based on ensuring that every rule r applied to a given state s_{before} yields a state s_{after} with e_{after} being smaller than e_{before}. An expression's length is given based on the length of its subexpressions. The arrays encoding mainly changes constraint generation, and in the cases where rules are slightly modified they generate resulting expressions of the same size, thus guaranteeing termination.

Soundness. This property is described in Theorem 1. Both e and Φ are KerA$^+$ expressions, but Φ is in the first-order logic fragment supported by SMT solvers. Fundamentally, the ARS is rewriting a formula to forward it to the solver. The soundness proof consists of case analysis of each reduction rule to establish that $e_{before} \wedge \Phi_{before}$ is equisatisfiable to $e_{after} \wedge \Phi_{after}$, no matter the rule applied in

$s_{before} \rightsquigarrow s_{after}$. The case analysis, which describes how e_{after} and Φ_{after} can be derived from e_{before} and Φ_{before} for each rule, relies on six invariants of the reduction system. Three invariants, 1, 3, and 4, are encoding independent, and thus are the same as those by Konnov et al. [2019], the remaining three, 2, 5, and 6, are changed due to the new representation of sets and functions. Below all six invariants are shown, with the modifications needed for the arrays encoding.

Theorem 1. Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_n$ be a sequence of states produced by the ARS, with $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ and $1 \leq i \leq n$. Assume that e_0 is a formula, i.e., it has type Bool. Then e_0 is satisfiable iff the conjunction $e_n \wedge \Phi_n$ is satisfiable.

Invariant 1 (type correctness)**.** In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, the expression e is well typed.

Invariant 2 (arena membership)**.** In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, every cell c in either the expression e or the formula Φ is also in $\mathcal{A}$.

Invariant 3 (model suitability)**.** Let $s_{before} \rightsquigarrow s_{after}$ be a reachable transition in the ARS, and $\mathcal{M}_{before}$ be a suitable model for s_{before}. An extended model $\mathcal{M}_{after}$ from $\mathcal{M}_{before}$ is suitable for s_{after}.

Invariant 4 (overapproximation)**.** Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_{set} is a set cell in the arena $\mathcal{A}$ and that $\mathrm{c}_{set} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n$ are edges in $\mathcal{A}$, for some $n \geq 0$. Then, it must hold that $[\![\mathrm{c}_{set}]\!]^{\mathcal{M}} \subseteq \{[\![\mathrm{c}_1]\!]^{\mathcal{M}}, \ldots, [\![\mathrm{c}_n]\!]^{\mathcal{M}}\}$.

Invariant 5 (function domain)**.** Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS. Assume that c_f is a function cell of type $\mathsf{s}_{\tau_1} \rightarrow \mathsf{s}_{\tau_2}$ in the arena $\mathcal{A}$. Then, there is a cell c_{dom} of type $\mathsf{s}_{\mathsf{Set}[\tau_1]}$ such that $\mathrm{c}_f \xrightarrow[\mathcal{A}]{1} \mathrm{c}_{dom}$.

Invariant 6 (domain reduction)**.** Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_f is a function cell and that $\mathrm{c}_f \xrightarrow{1} \mathrm{c}_{F_{dom}}$ is in the arena $\mathcal{A}$. Then, it follows that $[\![\mathrm{c}_{F_{dom}}]\!]^{\mathcal{M}} = [\![\textsc{domain} f]\!]^{\mathcal{M}}$.

As described in sections 3.2.1 and 3.2.2, arrays precisely model TLA$^+$ sets and functions. The handling of sets revolves around membership constraints of form $\mathrm{c}_{set}[\mathrm{c}_i]$, which and can be set to *true* or *false* via *store*. Regarding functions, function application and update are trivially equivalent to array access and update. The more elaborate array operators also have a counterpart in TLA$^+$. Constant arrays are equivalent to a function definition for which all range values are the same constant, and array map is equivalent to set map. These equivalences explain how the changes in the arrays encoding do not invalidate the case analysis of the reduction rules used to prove Theorem 1, thus guaranteeing soundness.

3.3 Implementation and Evaluation

In order to evaluate the performance impact of the arrays-based encoding, it was implemented in the APALACHE model checker, which currently supports the constants encoding. Given a TLA$^+$ specification containing a property P, APALACHE is capable of performing bounded model checking up to a length k and, if P is an inductive invariant, it can check if the property holds with an unbounded length. In both modes, APALACHE checks if the SMT formula encoding the specification is satisfiable when conjoined with $\neg P$, and if that is the case a CEX in the form of a trace is produced using the arena information and the satisfiable assignment provided by the SMT solver. The implementation adds new reduction rules to APALACHE, which can be enabled via a CLI flag. When enabled, these rules replace the existing ones encoding sets and functions, as described in Section 3.2. In addition, APALACHE's CEX generation was also extended to handle assignments to SMT formulas containing arrays. Z3 [de Moura and Bjørner, 2008b] was used as the back-end solver. APALACHE is open-source and freely available[1].

A number of experiments were performed using APALACHE and the explicit state model checker TLC. For APALACHE, both its existing constants encoding and two versions of the arrays encoding proposed, called *arrays* and *funArrays*, were evaluated. The *arrays* version encodes both TLA$^+$ sets and functions as arrays, while the *funArrays* version encodes only TLA$^+$ functions as arrays. The purpose of having two versions of the proposed encoding is to evaluate the impact of encoding sets and functions as arrays separately. The evaluation setup consisted of a machine with 64 AMD EPYC 7452 processors and 256 GB of memory. The benchmarks used are first presented and then the results obtained are discussed.

3.3.1 Benchmarks

The TLA$^+$ specifications of three asynchronous protocols are considered as benchmarks. The first benchmark is a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [1985], showed in Figure 2.1, refered to as *aba*. The second benchmark is a specification of the consensus algorithm with Byzantine faults in one communication step by Dobre and Suri [2006], referred to as *cab*. The third benchmark is a specification of the asynchronous non-blocking atomic commitment protocol by Guerraoui [2001], referred to as *nac*. The common use of *aba* and *cba* is in replication scenarios with $N = 3F + 1$ replica nodes to tolerate F failures, while the *nac* protocol is typically used for

[1] Available at `https://github.com/informalsystems/apalache`.

partitioned databases. All the specifications used are freely available online[2].

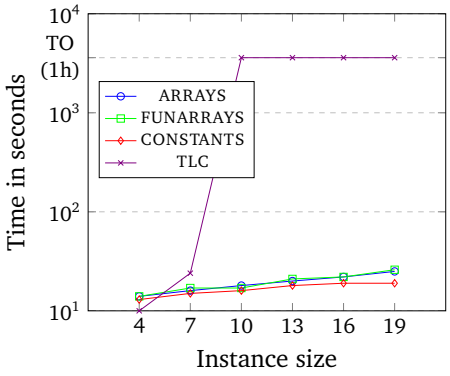
3.3.2 Results

For each specification the agreement property was checked. The results are shown in Figure 3.3. We can see that both *arrays* and *funArrays* scale in performance better that the constants encoding, with an order of magnitude improvement for some instances. It is also worth pointing out that *arrays* and *funArrays* were able to reach a result before the time limit in 29 and 28 instances, respectively, while the constants encoding was able to do so in only 20 instances. In regards to TLC, it performed worse than the three APALACHE encodings in the nontrivial cases, only reaching a result before the time limit in 8 instances.

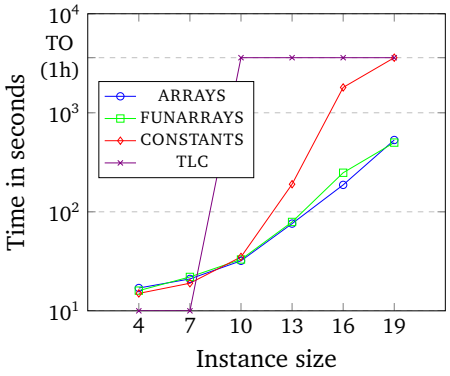
3.4 Related Work

An extensive discussion of works related to symbolic model checking for TLA$^+$ was done by Konnov et al. [2019]. Here the focus is exclusive to closely related publications. The IVy Prover [Padon et al., 2016] was designed to tackle verification of distributed algorithms with a decidable fragment of relational first-order logic. Some distributed algorithms, such as the one in Figure 2.1, cannot be directly expressed in this fragment however, due to the use of power sets and set cardinalities. Recent efforts have focused on offering support to reason about set cardinalities [Berkovits et al., 2019], but limitations remain. Cut-off based techniques to automatically infer invariants of distributed algorithms in the IVy language, such as relational abstractions of Paxos and two-phase commit, have been recently proposed [Goel and Sakallah, 2021a,b]. Similar benchmarks are used by Schultz et al. [2022] to infer generalized invariants from finite instances of TLA$^+$ and semi-automatically prove invariants with TLAPS. Specifications of fault-tolerant distributed algorithms encoded as threshold automata can be efficiently verified with ByMC [Konnov et al., 2020; Stoilkovska et al., 2022]. The manual rewriting of an algorithm into threshold automata is, however, usually beyond the skills of a typical TLA$^+$ user. The work closest to ours involves the use of SMT arrays to encode EventB and TLA$^+$ specifications in ProB [Schmidt and Leuschel, 2021]. The focus on ProB aims at handling infinite data structures, in contrast to our choice to work with bounded overapproximations. Reasoning about infinite domains implies the use of quantifiers, which prevents the use of efficient decision procedures available for the decidable fragment of SMT, with

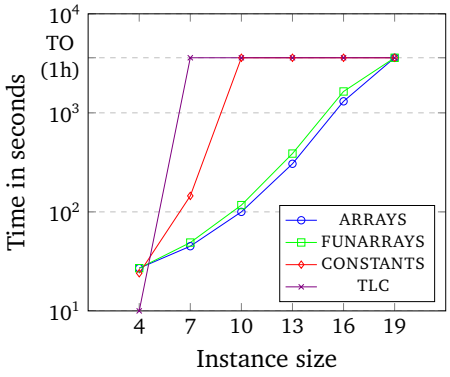
[2]Available at `https://github.com/informalsystems/apalache-bench`.



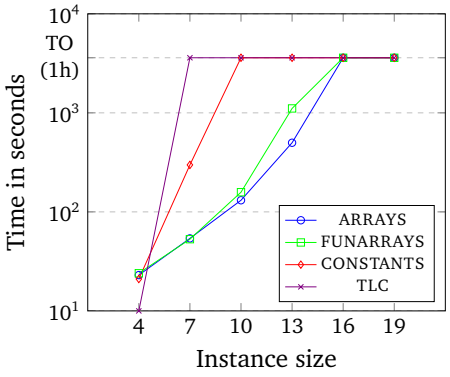
(a) Results for aba OK.



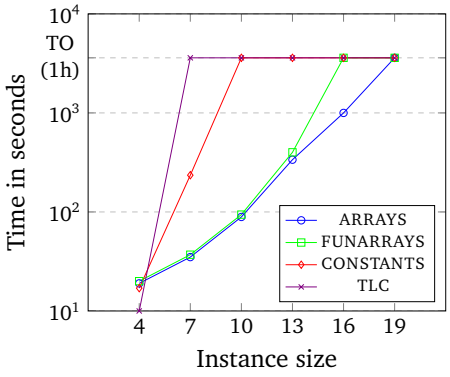
(b) Results for aba NotOK.



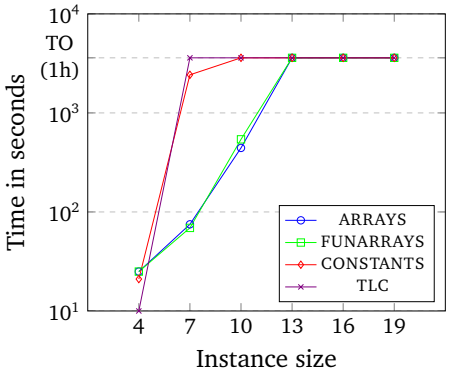
(c) Results for cab OK.



(d) Results for cab NotOK.



(e) Results for nac OK.



(f) Results for nac NotOK.

Figure 3.3. Time in checking agreement for aba, cab, and nac. The specifications were ran in two configurations, one in which agreement is expected to hold (OK) and one in which it is not (NotOK). Instance size stands for the number of nodes used, starting at 4 and being incremented by 3 up to 19, and the time is given in seconds in logarithmic scale; Timeout (TO) is 1 hour.

this approach been shown to underperform when compared against APALACHE in checking the benchmarks from the work of Konnov et al. [2019].

Two important last points regarding the handling of data structures are that non-standard SMT theories might be relevant in this context and that approaches not relying on SMT encoding are also possible. For the first point, a good example is CVC5's theory of sets [Bansal et al., 2016]. Despite focusing on sets, this theory cannot currently handle nested sets, which is a very commonly used TLA$^+$ construct. It remains as a viable alternative to the SMT theory of arrays for the encoding of flat sets, but whose use implies important restrictions to the input language and, consequentially, to practical application. For the second point, the axiomatic approach followed by BOOGIE [Barnett et al., 2006] illustrates an alternative way to handle structural information.

3.5 Conclusions and Future Work

In this chapter the novel encoding of the main TLA$^+$ constructs into the SMT theory of arrays proposed was presented, whose goal is to provide the SMT solver with the structural information it needs to efficiently reach a solution. The encoding was implemented into the APALACHE model checker and the evaluation indicates that it provides a significant performance improvement when compared against APALACHE's existing encoding and the explicit state model checker TLC.

Directions for future work include (i) encoding the remaining TLA$^+$ constructs in a structure-preserving way, (ii) evaluating the encoding with different back-end solvers, and (iii) developing an efficient unbounded model checking approach for TLA$^+$. For the first direction two possibilities are the continued use of arrays or, alternatively, the use of algebraic datatypes. For the second direction, candidate solvers are CVC5 [Barbosa et al., 2022a] and OPENSMT [Bruttomesso et al., 2010]. For the third direction, a CHC-based approach is considered as a suitable target.

Chapter 4

A Solicitous Approach to Smart Contract Verification

The safety-critical nature of smart contracts makes formally verifying them a necessity. The main programming language specifically designed to enable the implementation of smart contracts currently is Solidity, and this is the language in which efforts were focused. While existing works make use of different intermediary representations to reuse available verification tooling, Marescotti et al. [2020] proposed a direct encoding of Solidity into CHC in order to avoid the need of error-prone and potentially costly translations. This encoding enables the verification of assertions present in the source code, which are taken as the contracts' specifications. Assertion checking allows developers to not only ensure the absence of common known vulnerabilities, via injection of assertions that block attacking behaviours, but to also check functional correctness. By using CHC, once the set of contract states reachable is determined, it must be the case that either a contract invariant is established, proving that a given property holds after an unbounded number of transactions, or a finite-length CEX is found, concretely showing a property violation. Contract invariants are conditions over the contract's variables that always hold, which can thus be used by developers to confirm their intents for the code, while CEXs are lists of transactions that lead to an assertion error, which can aid in the fixing of vulnerabilities.

This chapter focuses on symbolic model checking for Solidity. The encoding by Marescotti et al. [2020] allows for unbounded model checking, meaning that vulnerabilities that happen after any finite sequence of transactions can be captured, but it produces constraints whose solving is undecidable in the general case. Despite this theoretical limitation, CHC solvers are able to reach a decision in many practical scenarios. To asses the practical use of CHC in smart

contract verification the tool implementing the encoding, SOLICITOUS, was extensively evaluated by executing it with 22446 real world Solidity contracts currently deployed on the Ethereum blockchain. In addition, a comparison was drawn against three publicly available tools suitable for automated verification of Solidity assertions, namely SRI's SOLC-VERIFY [Hajdu and Jovanovic, 2020], Microsoft's VERISOL [Wang et al., 2020], and ConsenSys' MYTHRIL [ConsenSys, 2021]. The results show that CHC-based model checking of smart contracts is viable in practice and that SOLICITOUS outperforms the compared tools. Finally, an extension of the encoding to improve CEX generation is also proposed.

4.1 Encoding Smart Contracts using CHC

The encoding by Marescotti et al. [2020] is presented in this section, together with the extension proposed to it. The encoding has a smart contract's CFG as its starting point, thus CFGs are described first. After that, the formal definition of a smart contract is presented, followed by the details regarding the modelling of contract's functions, static function calls, and dynamic function calls. The overarching algorithm used to create the CHC model of an entire contract is then presented, followed by the detailing of how safety can be checked, and, finally, how CEXs can be produced. The extension of the encoding consists of four new rules, namely $\mathsf{SumId}_{g,id}$, $\mathsf{Call}_{g,id,\rho_{call}}$, $\mathsf{ExtId}_{\mathbb{C},id}$, and $\mathsf{ECall}_{id,\rho_{call}}$, which allow function calls, both internal and external, to be better represented in CEXs.

4.1.1 Control-Flow Graphs

A control-flow graph is a graph representation of the execution paths of a program, commonly used for static analysis. The graph's nodes represent *basic blocks*, i.e., sequences of program statements that do not change the control flow of the program, while its edges represent *jumps* modelled after the program's control structures. Programming structures that modify the control flow include conditionals, loops, and function calls, with each edge in a CFG being labelled with a Boolean expression that must be true for the jump to occur.

Given that a smart contract's transaction is rooted at a call to one of its public functions, which can lead to additional function calls, a set of CFGs, each modelling one function, can accurately capture the contract's behaviour. In Figure 4.1 the CFG of function `offer` of contract `Auction` from Figure 2.2 can be seen. The **`assert`** statements are modelled as *safety blocks*, which contain direct jumps to the exit block guarded by the negation of the asserted expressions. The **`require`**

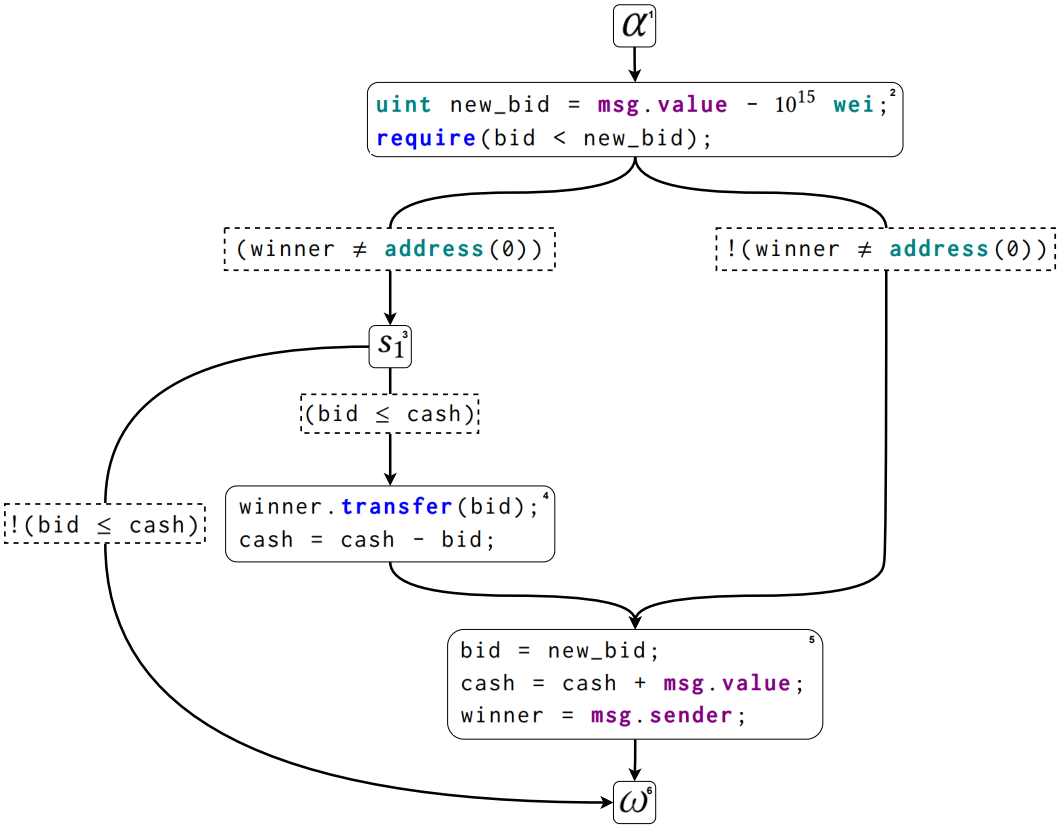


Figure 4.1. Graphical representation of the CFG of function `offer`. Solid and dashed rectangles represent blocks and jumps' conditions, respectively, and α, ω, and s_1 correspond to the entry, exit, and safety blocks.

statements are treated as execution constraints, i.e., they do not affect the control flow and only exclude invalid executions, which is their intended purpose.

In symbolic reasoning the state space of a program consists of the Cartesian product of the domain of the program counter and the domains of the program's variables, with the full space being restricted by writing predicates that describe the states reachable by the program. With this approach there is no need to list all the reachable states, thus the length of the search space representation in a logic is often vastly smaller than the explicit representation. In principle, the idea is to compute points of interest that a program can reach, given the its control flow and basic blocks in the static single assignment (SSA) form. Conceptually, each node in the CFG maps to a set of states. The initial states of the program can be obtained from constructors and other initialization. After this, the control flow is traversed, adding new states as we go to the sets associated with the nodes.

This traversal continues until a fixed point is reached with respect to the state sets. Finding logical formulas that represent these states is, however, not easy. To construct the formulas, recently developed methodology that combines well-defined semantics of this intuition with robust, albeit rapidly evolving, solving technology is applied.

4.1.2 Basic Definitions and Notation

A contract C is defined as a 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 the sets of public and private functions of F. For example, the formal definition of the contract `Auction` in Figure 2.2 is

$$Auction = \langle \{b, c, w\}, b = 0 \wedge c = 0 \wedge w = 0, \{offer(\{v, s\}) \rightarrow \{\tilde{r}\}\} \rangle$$

with b, c, and w representing the state variables `bid`, `cash`, and `winner`, v and s representing the implicit function arguments `msg.value` and `msg.sender`, $\tilde{r}$ being a special variable not derived from the source code, used to capture the occurrence of a revert, and $offer \in F^+$. In order to keep the example simple, we refrain from modelling implicit state variables, e.g., `balance` $\in \boldsymbol{s}$, which records the amount of funds held by the contract.

Given a function $f(\boldsymbol{a}) \rightarrow \boldsymbol{r} \in F$, where $\boldsymbol{a}$ is the set of function arguments, and $\boldsymbol{r}$ is the set of return variables, the CFG of function f is the tuple $\langle G, \alpha, \omega, \rho \rangle$. $G = (V, E, \lambda, \mu, S)$ is a node- and edge-labelled directed graph, where V is the set of CFG blocks, $E \subseteq V \times V$ is the set of control flow jumps, $\lambda_v \in \lambda$ is, for all $v \in V$, the set that contains the instructions performed by v, $\mu_e \in \mu$ 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, with each such block representing a safety property. The CFG blocks $\alpha, \omega \in V$ are respectively the entry and the exit blocks. The CFG of function `offer`, represented as CFG_o, is

$$\begin{aligned}
&\text{CFG}_\text{o} = \langle G_\text{o}, 1, 6, \rho_\text{o} \rangle \\
&\quad G_\text{o} = (\{1,2,3,4,5,6\}, \{\langle 1,2\rangle, \langle 2,3\rangle, \langle 2,5\rangle, \langle 3,4\rangle, \langle 3,6\rangle, \langle 4,5\rangle, \langle 5,6\rangle\}, \lambda, \mu, \{3\})
\end{aligned}$$

with the block identifiers in Figure 4.1 representing the graph's nodes and the instructions pertaining to λ and μ also following from the depiction in Figure 4.1, e.g., $\lambda_1 = \lambda_3 = \lambda_6 = \emptyset$ and $\mu_{(1,2)} = \mu_{(4,5)} = \mu_{(5,6)} = \texttt{true}$.

In the encoding of a function f only local variables are manipulated, therefore the labelings λ and μ of each block and jump contain instructions performed only

over a set of local variables $\boldsymbol{l}$ of f; changes to state variables are first done in local copies of them, and only committed if f terminates successfully. 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 mapping notation is extended to sets naturally: for a given set of variables $\boldsymbol{z}$, $\rho(\boldsymbol{z}) = \{\rho(x) \mid x \in \boldsymbol{z}\}$. The injection $\rho_\circ$ present in $\text{CFG}_\circ$ is

$$\rho_\circ = \{b \to l_b, c \to l_c, w \to l_w, v \to l_v, s \to l_s, \tilde{r} \to l_{\tilde{r}}\}$$

with l_x, $x \in \{b, c, w, v, s\}$, representing the local copy of variable x and $l_{\tilde{r}}$ representing the local copy of the revert variable.

A safety property in the CFG is represented by a safety block. In Solidity, safety properties are specified with the **`assert`** keyword, and their 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 the changes made by the function, restoring the state to prior the function's execution. In order to provide ω with the information that a safety property has been broken, λ_b sets $\rho(\tilde{r}) \in \boldsymbol{l}$ to a value that uniquely identifies the violated safety property, with $\tilde{r} \in \boldsymbol{r}$ being a special revert variable not derived from the source code. For the function `offer` we have one safety block, block 3, with $\mu_{(3,4)}$ being the property that is asserted, `bid` $\leq$ `cash` as shown in Figure 4.1, and $\mu_{(3,6)}$ being its negation.

4.1.3 Contract's Functions

Given a contract C with state variables $\boldsymbol{s}$, a function $f(\boldsymbol{a}) \to \boldsymbol{r} \in F$, with local variables $\boldsymbol{l}$ and CFG $\langle G, \alpha, \omega, \rho \rangle$, is modelled in the following manner. For each CFG block v, $\mathcal{P}^v_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l})$ is the predicate symbol that represents the states that are reachable in block v, and the SSA formula $\text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$, with $\boldsymbol{l}' = \{x' \mid x \in \boldsymbol{l}\}$, models the behaviour of v by formalising in first-order logic the relation between x and x', for each $x \in \boldsymbol{l}$, based on the execution of the instructions in λ_v. For each CFG jump e, the formula $\text{SSA}_{\mu_e}(\boldsymbol{l})$ is the logical condition under which e is taken. The execution of f is defined by three rules.

The *jump rule* models each jump $e = \langle v, u \rangle \in E$, and is expressed by the CHC

$$\mathcal{P}^u_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}') \leftarrow \mathcal{P}^v_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{l}) \wedge \text{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \wedge \text{SSA}_{\mu_e}(\boldsymbol{l}) \qquad (\text{Jump}_{f,e})$$

The function `offer` contains seven jumps, as seen in the definition of G_o and illustrated on Figure 4.1. Applying the rule $\mathsf{Jump}_{f,e}$ to these jumps yield the following CHCs:

$$\mathcal{P}_o^2 \leftarrow \mathcal{P}_o^1 \qquad (\mathsf{Jump}_{o,\langle 1,2\rangle})$$
$$\mathcal{P}_o^3 \leftarrow \mathcal{P}_o^2 \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge l_w \neq 0 \qquad (\mathsf{Jump}_{o,\langle 2,3\rangle})$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^2 \wedge l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb} \wedge \neg(l_w \neq 0) \qquad (\mathsf{Jump}_{o,\langle 2,5\rangle})$$
$$\mathcal{P}_o^4 \leftarrow \mathcal{P}_o^3 \wedge l_b \leq l_c \qquad (\mathsf{Jump}_{o,\langle 3,4\rangle})$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^3 \wedge \neg(l_b \leq l_c) \wedge l_{\tilde{r}} = 3 \qquad (\mathsf{Jump}_{o,\langle 3,6\rangle})$$
$$\mathcal{P}_o^5 \leftarrow \mathcal{P}_o^4 \wedge l'_c = l_c - l_b \qquad (\mathsf{Jump}_{o,\langle 4,5\rangle})$$
$$\mathcal{P}_o^6 \leftarrow \mathcal{P}_o^5 \wedge l'_b = l_{nb} \wedge l'_c = l_c + l_v \wedge l'_w = l_s \qquad (\mathsf{Jump}_{o,\langle 5,6\rangle})$$

with the signatures of $\mathcal{P}_o^1, \mathcal{P}_o^2, \mathcal{P}_o^3, \mathcal{P}_o^4, \mathcal{P}_o^5$, and $\mathcal{P}_o^6$, which represent the CFG blocks of function `offer`, being $(b, c, w, s, v, l_b, l_c, l_w, l_v, l_s, l_{nb}, l_{\tilde{r}})$. l_{nb} is used to represent the local variable `new_bid`. The **`transfer`** on line 11 of Figure 2.2 is abstracted out because `balance` $\in \boldsymbol{s}$ is not present. When a primed version of a variable appears in the body of a CHC, such variable is also primed in the head, e.g., in $\mathsf{Jump}_{o,\langle 5,6\rangle}$ we have $\mathcal{P}_o^5(b, c, w, s, v, l_b, l_c, l_w, l_v, l_s, l_{nb}, l_{\tilde{r}})$ representing all states reachable in block 5 and $\mathcal{P}_o^6(b, c, w, s, v, l'_b, l'_c, l'_w, l_v, l_s, l_{nb}, l_{\tilde{r}})$ representing the states reachable in block 6 by jump $\langle 5, 6\rangle$, which contains updates to l_b, l_c, and l_w.

To illustrate the application of $\mathsf{Jump}_{f,e}$ to one specific jump, let us consider the CHC $\mathsf{Jump}_{o,\langle 2,3\rangle}$. Its body contains predicate $\mathcal{P}_o^2$ conjoined with SSA_{λ_2}, which is the formula $l_{nb} = l_v - 10^{15} \wedge l_b < l_{nb}$ modelling the behaviour of block 2, and with $\mathrm{SSA}_{\mu_{\langle 2,3\rangle}}$, which is the formula $l_w \neq 0$ modelling the entry condition of the if-statement of function `offer`. Its head consists of the predicate $\mathcal{P}_o^3$, modelling the states in block 3 reachable through jump $\langle 2, 3\rangle$. For block 3, we can see that it is part of the body of two CHCs, $\mathsf{Jump}_{o,\langle 3,4\rangle}$ and $\mathsf{Jump}_{o,\langle 3,6\rangle}$, the former modelling normal execution and the latter modelling the case in which the assertion fails, with its head being the predicate encoding block 6, the exit block of `offer`.

The *entry rule* sets the local variables equal to the corresponding current values of state variables and passed arguments, and is expressed by the CHC

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

The variables in $\boldsymbol{s}$ and $\boldsymbol{a}$ are symbolically assigned in Entry_f and are never changed throughout the applications of $\mathsf{Jump}_{f,e}$, for 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 by a jump to ω setting

the local variable $\rho(\tilde{r})$ equal to the integer identifier of a safety property that failed, with $\rho(\tilde{r})$ being initially set to zero.

The CHC modelling the entry of function `offer`, produced by rule Entry_f, is the following:

$$\mathcal{P}_o^1 \leftarrow b = l_b \wedge c = l_c \wedge w = l_w \wedge s = l_s \wedge v = l_v \wedge l_{\tilde{r}} = 0 \qquad (\mathsf{Entry}_o)$$

The *function summary* of a given function is the relation between its input and output, derived from all possible function executions. Let $\mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r})$ be the predicate symbol representing the function summary of an execution of f. In this context, the input is represented by the state variables prior to the execution, $\boldsymbol{s}$, and the function arguments, $\boldsymbol{a}$, while the output is represented by the state variables after the execution, $\boldsymbol{s}'$, and the return values, $\boldsymbol{r}$. The *summary rule* is expressed by the CHC

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

Rule Sum_f contains three constraints, *revert* and *commit*, which are mutually exclusive, and *return*. The *revert* constraint 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. The *commit* constraint stores the local copy of the state variables in $\boldsymbol{s}'$, modelling a commit of the computed values. The *return* constraint equates the return variables $\boldsymbol{r}$ with their corresponding local variables.

The CHC modelling the summary of function `offer`, produced by rule Sum_f, is the following:

$$\mathcal{S}_o(b, c, w, v, s, b', c', w', \tilde{r}) \leftarrow \mathcal{P}_o^6 \wedge \big(l_{\tilde{r}} \neq 0 \implies b' = b \wedge c' = c \wedge w' = w\big) \quad (\mathsf{Sum}_o)$$
$$\wedge \big(l_{\tilde{r}} = 0 \implies b' = l_b \wedge c' = l_c \wedge w' = l_w\big) \wedge \tilde{r} = l_{\tilde{r}}$$

Definition 4.1.1 (Function Model)**.** Given a contract function f, the set of CHCs modelling f, Π_f, consists of applications of the jump rule $\mathsf{Jump}_{f,e}$, for each control flow jump e of f, and the entry and summary rules for f, Entry_f and Sum_f.

For function `offer` we have the set

$$\Pi_o = \{\mathsf{Jump}_{o,\langle 1,2\rangle}, \mathsf{Jump}_{o,\langle 2,3\rangle}, \mathsf{Jump}_{o,\langle 2,5\rangle}, \mathsf{Jump}_{o,\langle 3,4\rangle}, \mathsf{Jump}_{o,\langle 3,6\rangle}, \mathsf{Jump}_{o,\langle 4,5\rangle}, \mathsf{Jump}_{o,\langle 5,6\rangle}\} \cup \{\mathsf{Entry}_o, \mathsf{Sum}_o\}$$

4.1.4 Function Calls

Consider functions f and g, which need not be distinct, represented by CFGs G_f and G_g. A function call is performed by a block v in G_f if its labelling λ_v contains the call instruction to G_g. At runtime, the execution of the CFG block v is performed by executing the CFG block α of G_g, which constitutes the start of an execution of G_g. When ω of G_g is reached, the transaction represented by the execution of G_g is finalized by committing any changes to the state variables. The execution is then resumed from v, mapping the return variables of g to the corresponding 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. When the execution terminates, ρ is used to commit changes performed locally in the model to the state variables.

Consider a control flow jump $e = \langle v, u \rangle$, 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 by g after its execution terminates. To precisely represent distinct calls to the same function, in order to accurately record where an exception occurs, if it does so, uniquely tagged summaries are created for each call. These summaries are then used in the definition of their respective SSA formulas.

The tagged summaries are created by using the *summary id rule*, expressed by the CHC

$$\mathcal{S}_{g^{id}}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \leftarrow \mathcal{S}_g(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \qquad (\mathsf{SumId}_{g,id})$$

The first tagged summary of function `offer` would have been given by the following CHC:

$$\mathcal{S}_{o^1} \leftarrow \mathcal{S}_o \qquad (\mathsf{SumId}_{o,1})$$

with additional tagged summaries being possible through the CHCs $\mathsf{SumId}_{o,n}$, $n \in \{2, 3, ...\}$.

Given a freshly tagged summary, $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ of $\mathsf{Jump}_{f,e}$ containing a function call is defined as

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

The tagged summary maps s' to s'', instead of s to s', and appends the function's name to its arguments and return variables, in order to avoid a clash with the representation of the variables in the callee. The mapping $\rho_{call} : \boldsymbol{a}_g \to \boldsymbol{l}, \boldsymbol{r}_g \to \boldsymbol{l}'$ is specific to this call and maps both the arguments of g to $\boldsymbol{l}$ and the return variables of g to $\boldsymbol{l}'$. The set of local variables that can be affected by the call is $\boldsymbol{l}_{call} = \rho_{call}(\boldsymbol{r}_g) \cup \rho(\boldsymbol{s})$. The arguments are passed by value, thus local variables $\rho_{call}(\boldsymbol{a}_g)$ provided as arguments to function g are not affected by g's execution.

The SSA defined in $\mathsf{Call}_{g,id,\rho_{call}}$ constrains its function summary in a three-fold manner. First, the *arguments and returns passing* conjunction uses ρ_{call} to match arguments and return variables to the respective local variables of the caller. Second, the *state set and update* conjunction ensures that the local variables in $\boldsymbol{l}'$ that represent state variables are updated according to the execution g. Note that in case a revert occurs in the execution of g, $\rho(x) \equiv \rho(x)'$, meaning the state has not changed, and $\rho_{call}(\tilde{r_g}) \in \boldsymbol{l}$ is set according to the revert, which allows the modelling of revert propagation or catching by the caller. Lastly, for each local variable not in $\boldsymbol{l}_{call}$, the *untouched locals* conjunction equates its primed and non-primed versions, modelling that its value is not affected by the execution of g. Since all variables in $\boldsymbol{l}'$ are constrained, $\mathsf{Call}_{g,id,\rho_{call}}$ models a deterministic execution of the callee. By combining rules $\mathsf{Jump}_{f,e}$ and $\mathsf{Call}_{g,id,\rho_{call}}$, the resulting CHC is nonlinear, i.e., it contains more than one predicate in its body, in this case the predicates $\mathcal{P}^v_f$ and $\mathcal{S}_{g^{id}}$.

If the contract `Auction` contained a function `refundFee`, which allowed selected bidders to have their fees refunded, a call to it by some other function in `Auction` would be defined by the SSA

$$\begin{aligned}
&\mathcal{S}_{rf^1} \wedge v_{rf} = l_v \wedge s_{rf} = l_s \wedge \tilde{r}_{rf} = l'_{\tilde{r}} && (\mathsf{Call}_{rf,1,\rho_{call_{rf1}}})\\
&\quad \wedge (b' = l_b \wedge b'' = l'_b) \wedge (c' = l_c \wedge c'' = l'_c) \wedge (w' = l_w \wedge w'' = l'_w)\\
&\quad \wedge l'_v = l_v \wedge l'_s = l_s
\end{aligned}$$

with the summary $\mathcal{S}_{rf^1}$ given by the CHC

$$\mathcal{S}_{rf^1} \leftarrow \mathcal{S}_{rf} \qquad (\mathsf{SumId}_{rf,1})$$

The signatures of the predicates in $\mathsf{Call}_{rf,1,\rho_{call_{rf1}}}$ and $\mathsf{SumId}_{rf,1}$ are respectively $(b', c', w', v_{rf}, s_{rf}, b'', c'', w'', \tilde{r}_{rf})$ and $(b, c, w, s, v, b', c', w', \tilde{r})$. It is assumed that identifier 1 only tags this particular call to `refundFee`, and `refundFee` has neither explicit arguments and return variables nor local variables in its body.

4.1.5 Contract's External Behaviour

A transaction of a given a contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$ consists of the execution of a single public function of C. Contract transactions can therefore be modelled by the summaries of every function $f \in F^+$, with each summary providing the relation between the state variables before and after the execution of its associated function. The *external behaviour* of C encompasses all possible interactions between it and an external contract, being defined as the transitive closure of C's transactions. This way we capture the relation between the state variables before and after an arbitrary number of calls to any of the contract's public functions, in any order; to capture whether an assertion error occurs in such calls, the revert variable $\tilde{r}$ is also recorded. The predicate $\mathcal{E}_C$ that models the external behaviour of C is defined inductively. The *external base rule* is expressed by the CHC

$$\mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}, 0) \leftarrow \top \qquad (\mathsf{ExtBase}_{\mathcal{C}})$$

and the *external inductive rule* is expressed, for each $f \in F^+$, by the CHC

$$\mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \leftarrow \mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}', \tilde{r}') \wedge \mathcal{S}_f(\boldsymbol{s}', \boldsymbol{a}, \boldsymbol{s}'', \boldsymbol{r}) \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} \qquad (\mathsf{ExtInd}_{\mathcal{C},f})$$

The external behaviour of contract `Auction`, assuming it contains the function `offer` as well as the function `refundFee` described at the end of Section 4.1.4, is given by the following CHCs:

$$\mathcal{E}_A \leftarrow \top \qquad (\mathsf{ExtBase}_A)$$
$$\mathcal{E}_A \leftarrow \mathcal{E}_A \wedge \mathcal{S}_o \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} \qquad (\mathsf{ExtInd}_{A,o})$$
$$\mathcal{E}_A \leftarrow \mathcal{E}_A \wedge \mathcal{S}_{rf} \wedge \tilde{r}' = 0 \wedge \tilde{r}'' = \tilde{r} \qquad (\mathsf{ExtInd}_{A,rf})$$

with the signature of $\mathcal{E}_A$ being $(b^p, c^p, w^p, b^q, c^q, w^q, \tilde{r}^q)$, p and q being n primes, $n \in \{0, 1, 2\}$, depending of the rule being applied, and the signature of $\mathcal{S}_o$ and $\mathcal{S}_{rf}$ being $(b', c', w', s, v, b'', c'', w'', \tilde{r})$.

Predicate $\mathcal{E}_C$ can be used to model calls to functions of an external contract D, whose source code is unknown, capturing all possible interactions between C and D. Every control flow jump $\langle v, u \rangle$ of C in which block v contains a call to a function with unknown code is modelled by a uniquely tagged $\mathcal{E}_C$, instead of the called function's summary. The *external id rule* is given by the CHC

$$\mathcal{E}_{C^{id}}(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \leftarrow \mathcal{E}_C(\boldsymbol{s}, \boldsymbol{s}'', \tilde{r}'') \qquad (\mathsf{ExtId}_{\mathcal{C},id})$$

The first tagged predicate modelling the external behaviour of the `Auction`

contract is given by the following CHC:

$$\mathcal{E}_{A^1} \leftarrow \mathcal{E}_A \qquad (\mathsf{ExtId}_{A,1})$$

with $\mathcal{E}_{A^1}$ having the same signature as $\mathcal{E}_A$.

The SSA formula modelling the external call in block v, $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$, is similar to $\mathsf{Call}_{g,id,\rho_{call}}$, with two differences: the predicate used is $\mathcal{E}_{C^{id}}$, instead of $\mathcal{S}_{g^{id}}$, and the *arguments and returns passing* constraint is reduced to the update of $\rho(\tilde{r})'$. The local variables in $\rho_{call}(\boldsymbol{r}_g) \setminus \{\rho(\tilde{r})'\}$ are unconstrained in order to nondeterministically model any possible values returned by the unknown source code. The definition of $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}')$ is

$$\mathcal{E}_{C^{id}}(\boldsymbol{s}', \boldsymbol{s}'', \tilde{r}'') \wedge \underbrace{\rho(\tilde{r})' = \tilde{r}''}_{\textit{revert recording}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{s}} \left(x' = \rho(x) \wedge x'' = \rho(x)'\right)}_{\textit{state set and update}} \wedge \underbrace{\bigwedge_{x \in \boldsymbol{l} \setminus \boldsymbol{l}_{call}} x' = x}_{\textit{untouched locals}} \qquad (\mathsf{ECall}_{id,\rho_{call}})$$

If the function `refundFee` contained a call to an external function `unknown`, such a call would be defined by the following SSA formula:

$$\begin{aligned} &\mathcal{E}_{A^1} \wedge l'_{\tilde{r}} = \tilde{r}'' && (\mathsf{ECall}_{1,\rho_{call_{u1}}}) \\ &\quad \wedge (b' = l_b \wedge b'' = l'_b) \wedge (c' = l_c \wedge c'' = l'_c) \wedge (w' = l_w \wedge w'' = l'_w) \\ &\quad \wedge l'_v = l_v \wedge l'_s = l_s \end{aligned}$$

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. The existence of a CEX for this 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. The unique tag on $\mathcal{E}_{C^{id}}$ allows us to record which external function call led to the assertion failure, in case a CEX is produced, which is important because calls made at different points of a function's body can have different effects, in case state variables are being manipulated.

4.1.6 Contract's Complete Behaviour

Given a contract C, let $\mathcal{C}(\boldsymbol{s})$ be the predicate representing the reachable values for the contract's state variables. The contract's initial state is modelled by the

initialization rule, expressed by the CHC

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

Every root transaction with $f \in F^+$ is modelled by the *root transaction rule*, expressed by the CHC

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

For the contract `Auction` and function `offer` shown in Figure 2.2, we have

$$A \leftarrow b = 0 \wedge c = 0 \wedge w = 0 \qquad (\mathsf{Init}_A)$$
$$A' \leftarrow A \wedge \mathcal{S}_o \wedge \tilde{r} = 0 \qquad (\mathsf{RootTr}_{A,o})$$

Predicate A's signature is (b, c, w), with A' standing for $A(b', c', w')$; $\mathcal{S}_o$'s signature is as Sum_o's.

Definition 4.1.2 (Contract Model)**.** Given a contract C, the set of CHCs modelling all possible behaviours of C, Π_C, consists of applications of the initialization rule $\mathsf{Init}_{\mathcal{C}}$ and the external base rule $\mathsf{ExtBase}_{\mathcal{C}}$, together with the set of CHCs Π_f of every function $f \in F$ and, for each public function $f \in F^+$, the external inductive rule $\mathsf{ExtInd}_{\mathcal{C},f}$ and the root transaction rule $\mathsf{RootTr}_{\mathcal{C},f}$.

The complete CHC model of contract `Auction` is

$$\Pi_A = \{\mathsf{Init}_A, \mathsf{ExtBase}_A\} \cup \Pi_o \cup \{\mathsf{ExtInd}_{A,o}, \mathsf{RootTr}_{A,o}\}$$

The modelling technique is summarised in Algorithm 1. Given as input a smart contract C, the algorithm returns the set Π_C of CHCs modelling C. Initially, Π_C contains only applications of rules $\mathsf{Init}_{\mathcal{C}}$ and $\mathsf{ExtBase}_{\mathcal{C}}$. The loop from lines 1 to 26 iterates over every contract function f, constructing their respective Π_f sets, which are merged with Π_C in line 22, and applying rules $\mathsf{ExtInd}_{\mathcal{C},f}$ and $\mathsf{RootTr}_{\mathcal{C},f}$, in line 24, if the function is public. The internal loop from lines 6 to 21 iterates over every edge $e = \langle v, w \rangle$ of the CFG of f, in order to apply rule $\mathsf{Jump}_{f,e}$. For the modelling of block v, the case in which it represents a function call is handled in lines 7 to 15, using either the summary of the called function or the predicate of an external call, while if no function call is present in v, the formal model representing its execution is generated in line 17.

Input : contract $C = \langle \boldsymbol{s}, I(\boldsymbol{s}), F \rangle$
Output : set of CHCs Π_C
Initially: $\Pi_C \leftarrow \{\mathsf{Init}_{\mathcal{C}}, \mathsf{ExtBase}_{\mathcal{C}}\}$

foreach $f = \langle G, \alpha, \omega, \rho \rangle \in F$ with $G = \langle V, E, \lambda, \mu, S \rangle$ **do**
  $\boldsymbol{a} \leftarrow$ arguments of f
  $\boldsymbol{r} \leftarrow$ return variables of f
  $\boldsymbol{l} \leftarrow$ local variables of f
  $\Pi_f \leftarrow \{\mathsf{Entry}_f, \mathsf{Sum}_f\}$
  foreach $e = \langle v, w \rangle \in E$ **do**
    if v contains a call to $g(\boldsymbol{a}_g) \rightarrow \boldsymbol{r}_g$ **then**
      create ρ_{call} from λ_v *// maps arguments and returns of the call*
      if Sum_g is known **then**
        $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{SumId}_{g,id}\}$ *// adds a freshly tagged summary*
        $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathsf{Call}_{g,id,\rho_{call}}$
      else
        $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{ExtId}_{\mathcal{C},id}\}$ *// adds a freshly tagged external call*
        $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathsf{ECall}_{id,\rho_{call}}$
      end
    else
      $\mathrm{SSA}_{\lambda_v}(\boldsymbol{l}, \boldsymbol{l}') \leftarrow \mathrm{model}(\lambda_v)$ *// models according to v's instructions*
    end
    $\mathrm{SSA}_{\mu_e} \leftarrow \mathrm{model}(\mu_e)$ *// models according to e's conditions*
    $\Pi_f \leftarrow \Pi_f \cup \{\mathsf{Jump}_{f,e}\}$
  end
  $\Pi_C \leftarrow \Pi_C \cup \Pi_f$
  if $f \in F^+$ **then**
    $\Pi_C \leftarrow \Pi_C \cup \{\mathsf{ExtInd}_{\mathcal{C},f}, \mathsf{RootTr}_{\mathcal{C},f}\}$
  end
end

Algorithm 1: The algorithm to construct Π_C.

4.1.7 Checking Contract Safety

The safety of a public function $f \in F^+$, Σ_f, is modelled by the *safety rule*, expressed by the CHC

$$\bot \leftarrow \mathcal{C}(\boldsymbol{s}) \wedge \mathcal{S}_f(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s}', \boldsymbol{r}) \wedge \tilde{r} \neq 0 \qquad (\mathsf{Safety}_{\mathcal{C},f})$$

Definition 4.1.3 (Contract Safety)**.** Given a contract C, its safety condition consists of the set Σ_C containing the safety rules of every public function $f \in F^+$.

If $\Delta_\bot$ is empty for a given function f, then no transaction of f can result in an assertion error. A given contract C is safe if and only if the set of CHCs $\Pi_C \cup \Sigma_C$ is satisfiable. If an assertion error can be reached during the execution f, then there exist an interpretation of $\mathcal{C}$ and $\mathcal{S}_f$ that evaluates the body of the safety rule to *true*, making the set of CHCs unsatisfiable. Conversely, if the set $\Pi_C \cup \Sigma_C$ is satisfiable, then there exists a first-order formula $\eta(\boldsymbol{s})$ such that $\Delta_\mathcal{C} \models_T \mathcal{C}(\boldsymbol{s}) \implies \eta(\boldsymbol{s})$. The formula $\eta(\boldsymbol{s})$, called a *safe inductive invariant*, over-approximates the states reachable through any sequence of transactions and proves inductively that Σ_C always holds. Any formula $\iota(\boldsymbol{s})$ implied by a safe inductive invariant is a contract invariant, i.e., it is true in every reachable contract state.

For the code shown in Figure 2.2, having the predicates' signatures as in $\mathsf{RootTr}_{A,o}$, the safety rule of function `offer`, Σ_o, and the safety condition of contract `Auction`, Σ_A, are respectively

$$\bot \leftarrow A \wedge \mathcal{S}_o \wedge \tilde{r} \neq 0 \qquad (\Sigma_o)$$
$$\Sigma_A = \{\Sigma_o\}$$

Contract `Auction` has a potential vulnerability that can allow malicious users to siphon funds out of it, in a two step attack using function `offer`. First, the attacker makes a bid that is smaller than the bidding fee, e.g., by calling `offer` with `msg.value` $= 0$, leading to an underflow during the calculation of `new_bid` (line 7), which will be set to $2^{256} - 10^{15} +$ `msg.value`, assuming wrapping, i.e., overflow and underflow, is enabled; the type of `uint` is unsigned integer of 256 bits. This allows the attacker to set themselves as the winner, while sending no funds to the contract. Second, the attacker calls `offer` again, this time sending more funds than previously, but still less than the bidding fee, causing another underflow. Since the attacker is the current winner at this point, the function will refund them their previous bid (lines 9-13). The `assert` ensuring that a refund cannot be larger than the amount of funds gathered by the auction so far (line 10) prevents this attack.

A common practice in the domain of smart contracts is to add assertions during development, with the goal of catching unintended behaviours, but to remove them prior to deployment, in order to reduce deployment and execution costs. In the case of contract `Auction`, if the vulnerability was not caught prior to the removal of the `assert`, this would leave it open to the attack described, depending on the semantics of integer arithmetics adopted. Since the attack relies on

underflows happening, the satisfiability of the set $\Pi_A \cup \Sigma_A$ varies by language version. The newest Solidity version, v0.8, treats overflow and underflow as invalid operations by default, which automatically revert, making the attack impossible. Using v0.8 default semantics when generating the SSA formulas will thus make the set $\Pi_A \cup \Sigma_A$ satisfiable[1]. The Solidity versions preceding v0.8, which account for the overwhelming majority of deployed contracts, and that can be expected to be used for a significant number of contracts deployed in the near future as well, since developers tend to migrate slowly to newer versions, have, however, wrapping behaviour enabled, allowing the siphoning of funds.

By integrating the verification approach into the development process, developers can catch not only this simple vulnerability, but all vulnerabilities that can be specified using assertions, including documented exploits, and also check for functional properties. The checking procedure can easily be applied during development, since both the construction of the set of CHCs $\Pi_C \cup \Sigma_C$, for a given contract C, and the CHC solving, are fully automated, as detailed in Section 4.2.

4.1.8 Counterexample Production

The *refutation*, or proof of unsatisfiability, of $\Pi_C \cup \Sigma_C$ proves that a specific safety rule in Σ_C cannot be satisfied, i.e., $\Delta_\perp$ is nonempty. While the solving methodology can show satisfiability over unbounded executions, through the use of overapproximation, only finite refutations can be represented. This is, of course, not a practical limitation, since only vulnerabilities that manifest themselves after a finite number of steps are of interest. The description of how a refutation is constructed by the CHC solver is outside of the scope of this chapter, instead an overview of the refutations themselves and of how they are used in CEX production is given; proofs of unsatisfiability for CHC are touched upon in Chapter 6.

A refutation is a tree-shaped structure obtained by an unwinding of clauses. Its nodes are labelled with clauses, with its root, v_0, being labelled with a clause having $\perp$ as head. For each predicate $\mathcal{P}$ in the body of clause c of a given refutation node, a child labelled with a unique clause c' is created, with $head(c') = \mathcal{P}$. The leaves of the tree are labelled with clauses containing no predicates in their body. In a refutation, for all paths $v_0, \ldots, v_k$ from the root to a leaf, labelled with clauses $c_1, \ldots, c_k$, it must hold that

$$\models_T body_\phi(c_1)(\boldsymbol{x}_0, \boldsymbol{x}_1) \wedge body_\phi(c_2)(\boldsymbol{x}_1, \boldsymbol{x}_2) \wedge \ldots \wedge body_\phi(c_k)(\boldsymbol{x}_{k-1}, \boldsymbol{x}_k) \tag{DefRefPath}$$

[1] Solidity v0.8 allows wrapping behaviour via the `unchecked` command, which enables the semantics used in older versions.

with $body_{\phi}(c)$ denoting the constraint ϕ of a given clause c in the form DefClause.

A CEX is produced from a refutation by traversing the tree and listing the nodes that refer to the initialization rule $\mathsf{Init}_{\mathcal{C}}$, the root transaction rule $\mathsf{RootTr}_{\mathcal{C},f}$, and the safety rule $\mathsf{Safety}_{\mathcal{C},f}$. Due to the refutation's structure, traversing its left-most path gives us a list of nodes in which the first element is labelled with a safety rule, followed by zero or more elements labelled with root transaction rules, and the last element is a leaf labelled as an initialization rule. The conjunction of the clauses in the obtained list satisfies DefRefPath and represents a trace of transactions that leads to an assertion error, with the children of each node modelling the contract state prior the transaction and the function call with specific arguments that results in the new state.

Assuming underflow as a valid operation, $\Pi_A \cup \Sigma_A$ is unsatisfiable, and thus leads to a refutation, which is shown in Figure 4.2. In order to produce the CEX the tree is traversed, providing us with the list $\langle \Sigma_o, \mathsf{RootTr}_{A,o}, \mathsf{Init}_A \rangle$. The initial transaction in the CEX trace is the one from deployment, given by node Init_A, which sets `bid = 0`, `cash = 0`, and `winner = 0`, and is followed by the function calls modelled by the two other nodes. Node $\mathsf{RootTr}_{A,o}$ models the result of the first call to function `offer`, with `msg.value = 0` and `msg.sender = 0xA1`. Node Σ_o models the results of the second call to function `offer`, with `msg.value = 1` and `msg.sender = 0xA2`, which results in an assertion error.

4.2 Implementation

The approach was implemented inside the SOLCMC [Alt et al., 2022] component of the Solidity compiler SOLC[2], as part of a collaboration with the engineers of the Ethereum Foundation. The implementation, called SOLICITOUS (Solidity contract verification using constrained Horn clauses), consists of the CHC model checking engine of SOLCMC.

The functionality provided by SOLICITOUS can be enabled by simply adding `pragma experimental SMTChecker` directly into the source file, prior to the compilation. If enabled, the compiler provides the contract's CFG to SOLICITOUS, which produces the CHC model $\Pi_C \cup \Sigma_C$, following Algorithm 1 for Π_C and $\mathsf{Safety}_{\mathcal{C},f}$ for Σ_C. The CHC model is then provided to a CHC solver for checking, with SOLCMC currently supporting the SPACER [Komuravelli et al., 2016] and ELDARICA [Hojjat and Rümmer, 2018] solvers. In case an assertion error is detected, SOLICITOUS provides a transaction trace as a witness to the error, which can be easily validated by the developer. An overview of the SOLICITOUS architecture can be seen

[2]Available at `https://github.com/ethereum/solidity/releases`.

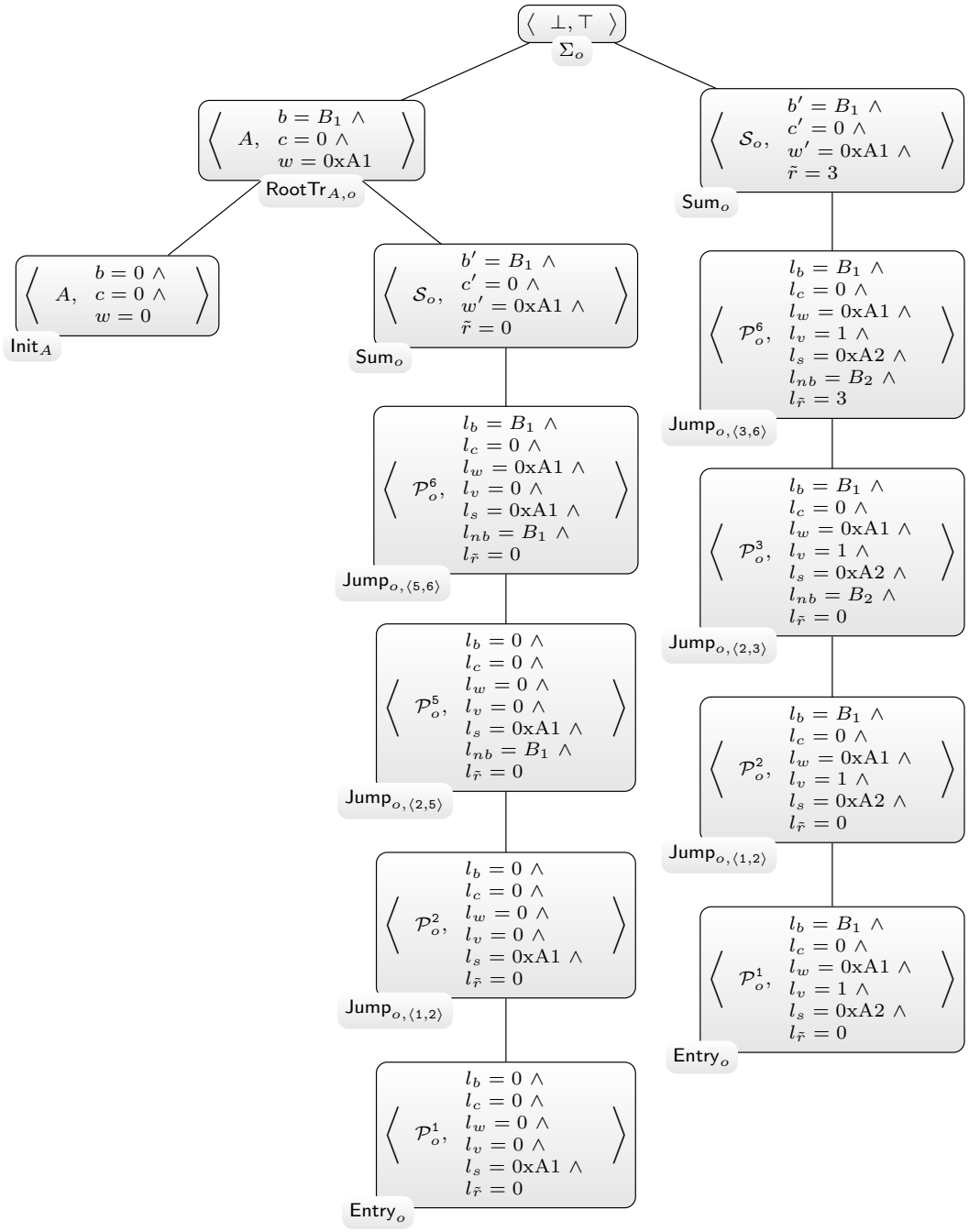


Figure 4.2. Refutation tree for the set $\Pi_A \cup \Sigma_A$. Besides its label, each node is annotated with a tuple containing the head of the referenced clause and the current value of relevant variables. The values 0xA1 and 0xA2 represent two Ethereum addresses, $B_1 = 2^{256} - 10^{15}$, and $B_2 = 2^{256} - 10^{15} + 1$.

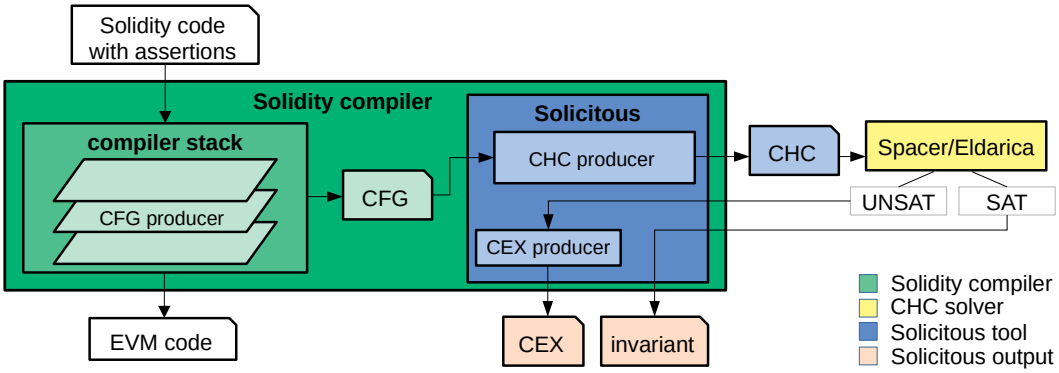


Figure 4.3. Solicitous module inside the Solidity compiler solc.

in Figure 4.3; the solver used, SPACER or ELDARICA, is treated as a black box.

The modelling of control flow structures such as conditionals and loops is not restricted by rule $\mathsf{Jump}_{f,e}$, following the topology of the CFG provided. The types of all variables in the CHC model directly reflects their source code equivalents, with addresses being treated as uninterpreted symbols and the unique names of the variables being derived from the CFG structure.

In addition to general functions, Solidity has two special function-like structures: *modifiers* and *constructors*. A modifier is a code fragment that envelops a number of selected functions' bodies. In SOLICITOUS, modifiers are not modelled separately, but are instead inlined to the functions they modify. A constructor contains the initialization procedure executed during the deployment of a contract. Constructors are used in the definition of $I(\boldsymbol{s})$, where variables are given either an explicit initial value or the default initial value of their type. In contracts with inheritance, the inheritance order is obtained by the compiler using C3 linearisation [Barrett et al., 1996], with each constructor being executed once. In SOLICITOUS, the deployment procedure, which might include state variables' initialization and inheritance linearisation, is inlined into a single `constructor`.

To the best of our knowledge, no verification tool targeting Solidity supports the complete range of language features, with all tools working on a best-effort basis. SOLICITOUS currently supports a large working subset of Solidity, including the complex control flow and arithmetic operators, excluding exponentiation, integers of all available sizes, Boolean variables, arrays, mappings' assignment and access, 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 in terms of both language features and versions is a goal of the Ethereum Foundation, with the supported subset of language expected to grow in the future.

4.3 Evaluation

Large-scale experiments were performed to evaluate both the precision and efficiency of the approach, as well as the current support of language features offered by the implementation. The evaluation was done in a fully automated fashion [Tange, 2011], enabling easy replication by interested third parties.

SOLICITOUS was compared against three publicly available tools suitable for automated verification of Solidity assertions: SRI's SOLC-VERIFY [Hajdu and Jovanovic, 2020] and Microsoft's VERISOL [Wang et al., 2020], that verify Solidity source code, and ConsenSys' MYTHRIL [ConsenSys, 2021], that verifies EVM bytecode. To the best of our knowledge these are the only tools with which an automated comparison is possible. Two other tools were considered for comparison, namely ZEUS [Kalra et al., 2018] and SAFEVM [Albert et al., 2019], but ZEUS is not publicly available and SAFEVM only supports Solidity v0.4.

Of the selected tools, SOLICITOUS, SOLC-VERIFY, and VERISOL can produce safe inductive invariants, and thus establish contract safety. MYTHRIL, however, is a purely bounded checking engine, being only capable of verifying up to a set number of transactions after contract deployment; by default a depth of three transactions is considered. Given this, MYTHRIL can never ensure safety, only report unsafe or inconclusive results. Despite these limitations, MYTHRIL is well known in the smart contracts community for having good support for language features, and it was chosen to serve as the gold standard for the language support metric. Regarding CEX production, SOLICITOUS can produce CEXs of arbitrary length, reporting assertion errors that can happen at any point during the lifecycle of a contract, while VERISOL and MYTHRIL can produce CEXs only up to given length, and SOLC-VERIFY does not produce CEXs at all; VERISOL has a hybrid approach, first attempting to establish unbounded safety, and if that is not successful it performs a bounded check.

The features implemented in SOLICITOUS vary by language version, with the description in Section 4.2 reflecting the current status of the v0.8 implementation. The support for language features is smaller for previous versions, with the v0.5 and v0.6 implementations not producing CEXs. This variation between versions is assumed to also hold for the other tools. SOLICITOUS supports the full range of Solidity versions from v0.5 to v0.8, as does MYTHRIL, but SOLC-VERIFY is restricted to v0.5, v0.7, and v0.8, while VERISOL only supports v0.5. Version 4.8.10 of Z3 [de Moura and Bjørner, 2008b], which contains SPACER, was used as the back end for all tools and the tools' encodings were set to modular arithmetic mode in order to properly capture the behaviour of arithmetic types.

4.3.1 Benchmarks

The benchmarks consist of real-world smart contracts deployed on the Ethereum blockchain over a period of 27 months. All contract deployment transactions present from block 7 million, mined on the 2nd of January 2019, to block 12 million, mined on the 8th of March 2021, were gathered and the Etherscan block explorer[3] queried for their respective source codes. The source code of 224186 contracts was obtained, of which 73614 were unique: 355 v0.8, 2617 v0.7, 16981 v0.6, 25306 v0.5, and 28355 of versions older than 0.5.

In the experiments only contracts containing `assert` statements were used, which were selected in two complementary ways. The first way was to simply select contracts already containing asserts, with these contracts being used as they were deployed. The second way involved contracts containing only asserts that were commented, with these contracts having their asserts uncommented. Commented asserts are of interest because developers might have commented them before deployment in order to reduce deployment and execution costs, believing them to hold. In total, 22446 contracts were used in the evaluation: 38 v0.8, including 61 asserts, 870 v0.7, including 1110 asserts, 9136 v0.6, including 11114 asserts, and 12402 v0.5, including 20912 asserts; older versions were not included due to lack of tool support. All benchmarks are publicly available[4].

4.3.2 Results

The summary of the results can be seen in Table 4.1, with the number of benchmarks available for each tool being derived from the Solidity versions it supports. A breakdown of results per Solidity version can be seen in Table 4.2. Safe contracts are those for which all asserts are proved safe by safe inductive invariants. Unsafe contracts are those for which a CEX was produced. Inconclusive results arise when the tool fails to either establish safety or produce a CEX. Timeout for each individual tool execution is 60 seconds. The difference between an inconclusive result and a timeout is that in the former the tool terminates successfully but is unable to classify the contract, while in the latter the process running the tool is killed upon reaching the time limit; in the case of a timeout a developer can run the tool for longer, which is not applicable for inconclusive results, arising due to fundamental limitations of the tool's approach. Crashes happen when language elements not handled by the tool are in the contract, e.g., inlined assembly code. A contract is considered verified if it is classified as safe or unsafe.

[3] See `https://etherscan.io`.

[4] Available at `https://scm.ti-edu.ch/repogit/verify-solidity-contracts.git`.

Table 4.1. Summary of experimental results. The best results are highlighted.

	SOLICITOUS (SOL)	SOLC-VERIFY (SV)	VERISOL (VS)	MYTHRIL (M)
Benchmarks	**22446**	13310	12402	**22446**
Safe	**6651**	103	201	0
Unsafe	8	0	9	**21**
Inconclusive	2970	5601	230	728
Timeout	9058	3163	4032	19511
Tool crash	3759	4443	7930	2186
Verified	**∼29%**	∼0.7%	∼1%	∼0.09%

SOLICITOUS was able to guarantee one order of magnitude more contracts to be safe, in comparison with SOLC-VERIFY and VERISOL. Regarding catching assertion errors, SOLICITOUS was the most performant tool for the versions in which it produces CEXs, v0.7 and v0.8, with MYTHRIL having the best result overall, probably due to the percentage of version 0.5 and 0.6 instances. The large number of benchmarks that lead to an inconclusive result or a timeout, among all tools, indicates the highly nontrivial nature of smart contracts verification. Regarding crashes, MYTHRIL has shown itself to be the more stable tool, as expected, with SOLICITOUS having the least crashes among the unbounded verification tools.

To compare the performance of the tools the runtimes of the executions that produced safe inductive invariants were gathered. The results are summarised in Figure 4.4. SOLICITOUS was able to verify more than 4000 contracts in less than 10 seconds, and more than 6000 in less than 30 seconds, which highlights its applicability for contract developers. SOLC-VERIFY's runtimes are also distributed throughout the time axis, with most of the 103 contracts classified as safe being done so in less than 40 seconds. VERISOL achieved all its 201 safe results in less than 10 seconds. The fact that the majority of the contracts were classified as safe in less than half the allocated time indicates that, for all tools, practical results can be achieved with small timeouts. We can also conclude that increasing the timeout can be beneficial for both SOLICITOUS and SOLC-VERIFY, but may not be for VERISOL. Given the positive results, aligned with the practical nature of the benchmarks, SOLICITOUS stands as a valuable tool for developers.

4.3.3 Manual Inspection and Vulnerabilities Found

To understand the types of vulnerabilities found by each tool and the contracts in which they occur, all thirty-eight contracts that were classified as unsafe in the

Table 4.2. Experimental results detailed per Solidity version. SOL, SV, VS, and M stand, respectively, for Solicitous, Solc-Verify, VeriSol, and Mythril. A dash means that the corresponding tool does not support the specific Solidity version. The best results are highlighted.

	SOL	SV	VS	M
Safe	**3689**	95	201	0
Unsafe	0	0	9	**13**
Inconclusive	2383	5327	230	279
Timeout	4101	3045	4032	9935
Tool crash	2229	3935	7930	2175
Verified	**∼29%**	∼0.7%	∼1%	∼0.1%

(a) Version 0.5, totalling 12402 instances.

	SOL	SV	VS	M
Safe	**2877**	-	-	0
Unsafe	0	-	-	**7**
Inconclusive	583	-	-	413
Timeout	4159	-	-	8706
Tool crash	1517	-	-	10
Verified	**∼31%**	-	-	∼0.07%

(b) Version 0.6, totalling 9136 instances.

	SOL	SV	VS	M
Safe	**72**	8	-	0
Unsafe	**7**	0	-	1
Inconclusive	4	268	-	35
Timeout	775	118	-	833
Tool crash	12	476	-	1
Verified	**∼9%**	∼0.9%	-	∼0.1%

(c) Version 0.7, totalling 870 instances.

	SOL	SV	VS	M
Safe	**13**	0	-	0
Unsafe	**1**	0	-	0
Inconclusive	0	6	-	1
Timeout	23	0	-	37
Tool crash	1	32	-	0
Verified	**∼36%**	0%	-	0%

(d) Version 0.8, totalling 38 instances.

experiments were manually inspected. In addition, all v0.8 contracts classified as safe by SOLICITOUS were also inspected, in order to understand how complex are the contracts for which SOLICITOUS can ensure safety.

Of the eight contracts classified as unsafe by SOLICITOUS, five are governance contracts based on ERC20, one is a token exchange contract based on ERC20 and ERC165, one is token sale contract, and one is a voting contract. These contracts have 584 lines of code on average, with the smallest having 97 lines of code and the largest having 1325 lines of code. The vulnerabilities found are caused by reentrant calls, affecting one contract, overflow, also affecting one contract, and unexpected inputs, affecting the remaining six contracts. Regarding the assertion failures caused by unexpected inputs, one is simply due to using `assert` for input validation instead of `require`, one is a contract owner call to a function containing the `selfdestruct` statement, which is guarded by `assert(balance > 0)`, with the implicit assumption being that the owner will transfer all funds from the contract before making such a call, and four are asserting properties on values returned from calls to external contracts. The vulnerabilities caused by unexpected inputs,

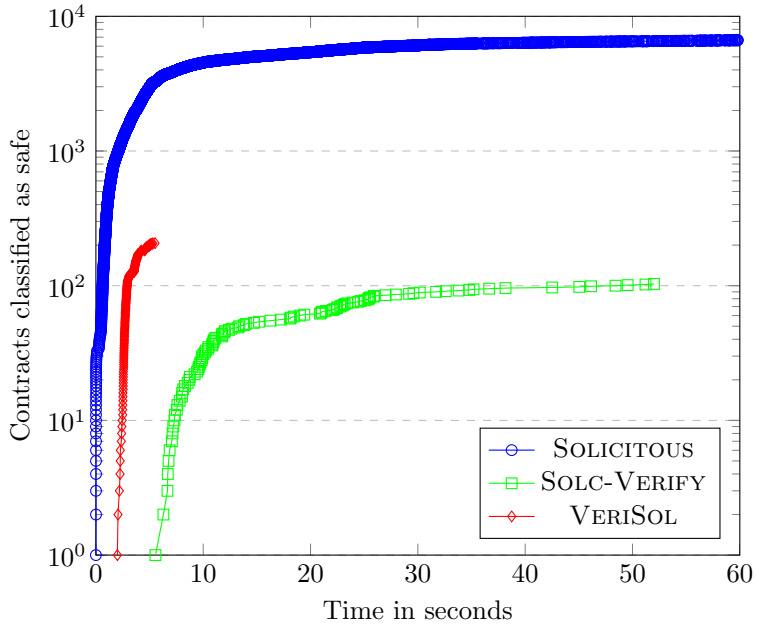


Figure 4.4. Number of contracts classified as safe by allocated time.

although less severe, are still problematic, since they lead to execution fees not being refunded depending on the Solidity version being used.

Of the nine contracts classified as unsafe by VERISOL, seven are governance contracts based on ERC20, one is a token sale contract, and one is a simple token transfer contract. These contracts have 229 lines of code on average, with the smallest having 37 lines of code and the largest having 397 lines of code. The vulnerabilities found are caused by overflow, affecting five contracts, unexpected inputs, affecting two contracts, and asserting a `transfer` statement, also affecting two contracts. Both assertions failures caused by unexpected inputs are due to using `assert` for input validation instead of `require`. Funds transferring can fail for various reasons pertaining to the target address and should thus be avoided.

Of the twenty-one contracts classified as unsafe by MYTHRIL, four are governance contracts based on ERC20, four are token sale contracts, two are token time lock contracts, four are wallet management contracts, two are airdrop contracts, three are logging contracts, which store hashes associated with timestamps, one is a signature contract, and one is a simple contract that delegates calls to a specified address. These contracts have 390 lines of code on average, with the smallest having 23 lines of code and the largest having 990 lines of code. The vulnerabilities found are caused by overflow, affecting thirteen contracts, unexpected inputs, affecting seven contracts, and unexpected branch execution, af-

fecting one contract. The assertion failures caused by unexpected inputs are due to validation of functions' inputs, two occurrences, and owner privileges checking, five occurrences. The assertion failure caused by an unexpected branch execution consists of an `assert(false)` statement in a part of the code that should not be reachable.

Of the thirteen v0.8 contracts classified as safe by SOLICITOUS, nine are governance contracts based on ERC20, two are token exchange contracts based on ERC20, ERC165, and ERC1363, one is a betting contract, and one is a vesting contract. These contracts have 565 lines of code on average, with the smallest having 233 lines of code and the largest having 721 lines of code. An interesting point is that three of those contracts contain comments stating that they were independently audited.

4.4 Related Work

There is much interest in formally verifying smart contracts. With Ethereum being one of the most used platforms for smart contract applications currently, most verification works target either EVM bytecode, which is deployed directly on the blockchain, or Solidity source code, which is compiled to EVM bytecode. Verification approaches vary in both their scope and the manner in which they are carried out. The scope ranges from the checking of specific vulnerabilities, usually selected from among high-profile documented exploits, to the use of different forms of specification languages, be they code assertions, design patterns, or behavioural descriptions. The manner can be either fully automated verification, aimed at developers, or manually intensive checking, usually intended to be used as auditing aid.

Automated Verification of Specific Vulnerabilities

OYENTE [Luu et al., 2016] is one of the pioneers in the field, using symbolic execution of EVM bytecode, underpinned by SMT solving, to check for common pre-defined vulnerabilities. MAIAN [Nikolić et al., 2018] uses symbolic execution to check for specific types of trace vulnerabilities in EVM bytecode, i.e., vulnerabilities that manifest themselves over many transactions, and also relies on SMT solving. MANTICORE [Mossberg et al., 2019] is based on a symbolic execution engine for EVM that searches for pre-defined vulnerabilities using SMT solving. SLITHER [Feist et al., 2019] is a static analysis framework for the verification of Solidity contracts containing several vulnerability detectors based on bounded

model checking. ETHAINTER [Brent et al., 2020] is a static analysis tool focused on tainted information detection on EVM bytecode, it relies on Datalog and can catch vulnerabilities such as free access to a contract's `selfdestruct` instruction. VERISMART [So et al., 2020] follows a counter-example guided inductive synthesis approach to verify arithmetic properties. Safety of arithmetic operations is also the target of SOLID [Tan et al., 2022], which uses a novel type system to achieve its goal. The issue of smart contract gas consumption, i.e., the payment of fees for the execution of contracts, and its related vulnerabilities, is considered by Marescotti et al. [2018], who use SMT solving as a means to estimate gas consumption. An approach aimed at preventing vulnerabilities from being introduced in the first is place is that of SCILLA [Sergey et al., 2019], an alternative programming language based on System F [Reynolds, 1974] built with contract safety as a principal concern. SCILLA's type system prevents a number of runtime vulnerabilities from being implemented and the language is accompanied by a framework for static analyses capable of checking specific properties, including the estimation of gas consumption. In contrast to previously mentioned works, however, SCILLA targets Zilliqa [Zilliqa Team, 2017] instead of Ethereum.

Compared to the approach described in this chapter, these techniques are restricted in terms of the scope of their verification. SOLICITOUS differs from them by not constraining the verification to predefined properties, allowing developers instead to check any properties that can be expressed via code assertions.

Automated Verification using Specification Languages

A verification framework based on F* [Swamy et al., 2016] as an intermediary language, enabling the translation of both EVM and Solidity code to F* and the subsequent checking of error patterns via SMT solving, is proposed by Bhargavan et al. [2016]. It allows for the definition of different patterns in F*, but lacks support for many important language features, such as loops. SECURIFY [Tsankov et al., 2018] encodes EVM bytecode into Datalog and checks for defined bytecode patterns, with a set of patterns being pre-defined and a specification language being provided for the definition of additional ones. ETHBMC [Frank et al., 2020] is a bounded model checker based on SMT solving targeting EVM bytecode. It can define a precise model of a contract's memory and is capable of checking a set of relevant properties, including access to the `selfdestruct` instruction. Extension of the checking capabilities is possible by encoding additional properties as constraints to be checked. ZEUS [Kalra et al., 2018] is a framework to check the correctness and fairness of smart contracts targeting the Ethereum or Fabric [Androulaki et al., 2018] blockchain platforms, with a fairness specification

language being defined, from which assertions are injected into the source code prior to verification. It translates high-level source code into LLVM [Lattner and Adve, 2004] bitcode, from which CHC are generated and discharged to a solver, using either the SEAHORN [Gurfinkel et al., 2015] or SMACK [Carter et al., 2016] model checkers. SMARTACE [Wesley et al., 2022] also translates high-level source code into LLVM, from which point it can use the aforementioned SEAHORN model checker, as well as the KLEE [Cadar et al., 2008] symbolic execution engine and the LIBFUZZER [LLVM Team, 2017] fuzzer. It is capable of checking assertions and Scribble annotations. SAFEVM [Albert et al., 2019] translates EVM bytecode to C, and then uses C verifiers to check properties such as invalid array access and division by zero, as well as assertion violations. ETHOR [Schneidewind et al., 2020] is a static analysis tool targeting EVM bytecode. It abstracts the bytecode into CHC via its HoRSt framework and uses reachability checking to verify the absence of vulnerabilities such as reentrancy. It can also check assertion failures by verifying the reachability of the `INVALID` EVM instruction. SMARTPULSE [Stephens et al., 2021] allows for the checking of safety and liveness properties in Solidity contracts. The desired properties need to be specified in the SmartLTL language and be provided to SMARTPULSE together with the contract's source code and a model of the environment in which it is expected to operate. SMARTPULSE instruments the contract's code based on the properties specified, translates it to the BOOGIE intermediary language, and then performs verification based on counter-example guided abstraction refinement.

The three tools used for comparison in the evaluation also fall in this category. SOLC-VERIFY [Hajdu and Jovanovic, 2020] translates Solidity source code to the BOOGIE intermediary language and then generates verification conditions that can be discharged to SMT solvers. In addition to checking for issues such as overflow and underflow, and assertion violations, it also provides support for code annotations that can complement assertions. VERISOL [Wang et al., 2020] also translates Solidity source code to the BOOGIE intermediary language, checking assertion violations first in an unbounded manner and, if that does not yield a result, in a bounded fashion. MYTHRIL [ConsenSys, 2021] is a tool capable of symbolic execution of EVM bytecode to check assertion violations and some specific properties. It relies solely on bounded analysis over a number of transactions, leaving undisclosed bugs that happen only after extended contract usage. In contrast to this chapter's approach, these techniques tend to rely on existing frameworks, e.g., BOOGIE, and provide weaker guarantees, e.g., MYTHRIL.

The fundamental problem that all cited tools attempt to tackle, that of verifying that a contract complies with a user-made specification, is also the target of SOLICITOUS. The difference here comes from how the problem is approached.

The cited works all rely on indirect representation in one way or another, even when CHC are involved, as is the case of ZEUS and ETHOR, while SOLICITOUS models the contracts directly in the formalism suitable for solving.

Manually Supported Verification

K [Roşu and Şerbănuţă, 2010] is a semantic framework that has specific support for EVM [Hildenbrandt et al., 2018], as well as Solidity [K Framework, 2018a] and Vyper [K Framework, 2018b], but is time-consuming and difficult for non-expert users to interact with, since it relies on manual intervention for verification. Deductive verification using WHY3 is proposed by Nehaï and Bobot [2020]. Their approach involves crafting and verifying smart contracts in Why3's language, WhyML, and then compiling them to EVM, but it was evaluated only on a single case study. VERX [Permenev et al., 2020] verifies functional properties of Solidity contracts written using its own specification language, it uses SMT solving and has a certain degree of push-button automation, but may require user input during the verification. The cited tools differ fundamentally from SOLICITOUS by requiring human intervention during the verification process, in addition to the specification of properties. Compared to the approach described in this chapter, these techniques have the obvious drawback of not being fully automated, with their target audience comprising mainly of highly specialized users, and should thus be considered orthogonal.

Usage of Constrained Horn Clauses in Other Domains

There have been many successful uses of CHC for verification in other domains. SEAHORN [Gurfinkel et al., 2015] has been used to verify programs targetting LLVM, prominently those written in C/C++, with one important feature being that it is modular w.r.t. the encoding, allowing, for instance, for encodings that consider or that abstract features such as memory usage. JAYHORN [Kahsai et al., 2016] and RUSTHORN [Matsushita et al., 2021] target programs written in Java and Rust, while HORNDROID [Calzavara et al., 2016] targets Android applications, with each tool catering for the different specificities of its target.

4.5 Conclusions and Future Work

In this chapter an approach for the automated verification of smart contracts based on direct modelling, which allows us to bypass intermediary steps com-

monly found in current verification approaches, was presented. The approach is instantiated to the Solidity language and targets the CHC formalism, leading to CHC models that (a) formally capture the semantic features specific of smart contracts, (b) enable efficient fully automated verification of contracts' properties, and (c) can directly exploit powerful CHC solvers for the production of both safe inductive invariants and CEXs. The approach was implemented in SOLICITOUS, the CHC model checking engine of the Solidity compiler's formal verification module SOLCMC. An extensive evaluation involving 22446 real-world contracts specifying in total 33197 properties was performed, comparing SOLICITOUS against three state-of-the-art tools. The results obtained demonstrate the benefits of the approach, with an order of magnitude improvement in the percentage of verified contracts. In light of the evaluation, it is believed that the approach represents an effective and highly promising avenue for the verification of smart contracts.

Directions for future work include (i) the further enhancement of the modelling, (ii) validation of the verification, and (iii) instantiation to other languages. For the first direction, enhancements could be made by considering both known external code during the verification, e.g., a call to a previously deployed contract written by the same development team, in order to increase precision, and the gas consumption of the contract's functions, to predict their execution fees. For the second direction, lightweight correctness certificates that can be used to independently and automatically validate the safety of smart contracts are envisioned, with their practical goal being to provide assurances to third parties about the contracts they interact with. Concretely, such certificates would validate the results of CHC solvers in a similar manner to what is already done with solvers for SAT and SMT, as discussed in the two following chapters. For the third direction, the approach could be instantiated to other languages for smart contract development to investigate its generality and extend its practical application.

One last point of interest is the relation between the formal semantics of Solidity and verification approaches, since an understanding of the language semantics is a necessity when providing correctness guarantees. To the best of our knowledge, no official semantics currently exists for Solidity, probably due to fast pace in which the language is evolving. Suggestions have been put forward by independent researchers, including formalisations based on the K framework [Jiao et al., 2020] and on ISABELLE/HOL [Marmsoler and Brucker, 2021]. It is unclear, however, how to best handle language semantics, besides closely following updates to the language documentation.

Chapter 5

Theory-Specific Proof Steps Witnessing Correctness of SMT Executions

Automated reasoning engines for FOL are a core component of a vast range of different approaches for symbolic model checking. The aim of these approaches is to leverage techniques from computational logic to efficiently traverse immense search spaces in order to determine whether hardware or software implementations conform to their specifications. Many model checkers ultimately reduce the verification problem to satisfiability queries in FOL expressed in a form suitable for SMT solvers [Kroening and Strichman, 2016]. As this becomes more common, the correctness of solver reasoning becomes critical. Efficient reasoning algorithms are, however, nontrivial to implement and might themselves contain bugs, as is often uncovered during the annual SMT competition. Some SMT solvers, e.g., Z3 [de Moura and Bjørner, 2008a], CVC4 [Katz et al., 2016], and VERIT [Besson et al., 2011], offer the capability of producing proofs in a given proof system [Barrett et al., 2015], that, among other uses, can partially respond to the need of increasing the trust in SMT solvers. The idea is that the proofs can be replayed with an external checker, which initially were mainly theorem provers such as COQ [Armand et al., 2011; Ekici et al., 2017] and ISABELLE/HOL [Böhme and Weber, 2010; Blanchette et al., 2016; Barbosa et al., 2017], that may accept or reject the proofs.

This chapter focuses on SMT unsatisfiability proofs that are compact and can be efficiently checked by lightweight checkers. The goal is to make it simple for interested parties, be they end users or model checkers, to ensure the correctness of results. The mathematical and logical foundations of the SMT algorithms were used to produce proof certificates that are simple enough to allow an auditor to write checkers for them with little effort, e.g., in a matter of hours or days. By

aiming at simplicity, proofs were able to be produced that are more compact on average than those of existing formats and that can be efficiently checked.

5.1 State of the Art

In this chapter the concerns on the correctness of SMT solvers previously identified by Van Gelder [2012] for solvers for Boolean satisfiability are addressed. The annual SAT competition requires competitors to produce proofs for ensuring the correctness of unsatisfiability results since its 2013 edition [Balint et al., 2013], and since 2014 it has adopted the deletion RAT (DRAT) format [Wetzler et al., 2014] as its standard proof format. DRAT is based on the property of SAT solvers that inserted clauses result in contradiction as a result of a limited polynomial-time computation, called *unit propagation*. The format is designed to express most current SAT solving techniques compactly and is a generalisation of the deletion reverse unit propagation (DRUP) [Heule et al., 2013a] and the resolution asymmetric tautology (RAT) [Heule et al., 2013b] formats. A variant, linear RAT (LRAT) [Cruz-Filipe et al., 2017], allows proof-checking in strictly linear time by decorating the DRAT format with indices serving as hints for unit propagation, with the price of an additional logging overhead during the search. While there is no widely accepted format for SMT proofs [Barrett et al., 2015], a subset of the DRAT format is used for the underlying SAT reasoning in the format proposed in this chapter.

The approach closest to the one proposed in this chapter is that of the SMT solver CVC4 [Barrett et al., 2011], which produces proofs in the LFSC metalogic [Stump et al., 2013] that can be checked by the automated LFSC checker. Similar to us, this system uses DRAT to validate the steps of the back-end SAT solver. The DRAT proofs, however, need to be translated into LFSC, creating a bottleneck both for proof production and for proof checking [Ozdemir et al., 2019]. CVC4's proofs can further be reconstructed in Coq [Ekici et al., 2017]. The veriT SMT solver [Bouton et al., 2009] is specifically designed for producing proofs and its proof format is being developed alongside the solver itself [Besson et al., 2011]. Unlike in our system, there was, until very recently, no independent checker for veriT proofs[1], but instead proofs could only be checked by using interactive theorem provers. A common workflow for SMT is to tune and replay a solver-specific proof in an interactive theorem prover such as Coq [Armand et al., 2011] and Isabelle/HOL [Blanchette et al., 2016]. The SMT solver Z3 [de Moura and Bjørner, 2008b] produces proofs in its own

[1]The new checker Carcara [Andreotti et al., 2023] is discussed in Chapter 6.

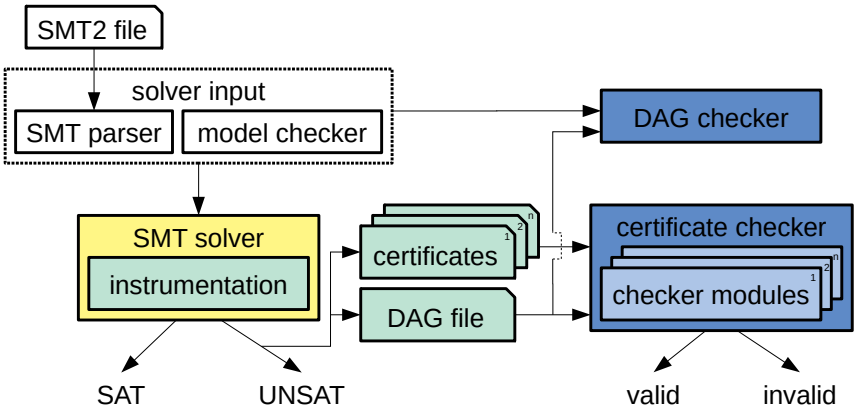


Figure 5.1. Overview of the system. The SMT solver and the artefacts related to certificate production and checking are shown in yellow, green and blue.

proof format [de Moura and Bjørner, 2008a], which can be reconstructed in Isabelle/HOL [Böhme and Weber, 2010]. Currently, Z3 is capable of producing proofs, but to the best of our knowledge no independent automated checker exists that can check proofs in its format.

5.2 SMT Unsatisfiability Proofs

Deriving new clauses from current ones is a key aspect of solving procedures, with this derivation happening in one of two ways: either as learned clauses, through resolution on previous clauses, or as theory clauses, when a theory solver reports that a set of theory (in)equalities is unsatisfiable. One way to obtain a proof certificate for the correctness of an SMT execution determining unsatisfiability is thus to produce partial certificates connecting an input formula F_{DAG} to the step in a propositional proof system deriving *false*. This includes proving the derivation of the learned clauses, the theory clauses, and the transformation into F_{CNF}. In this section each step is covered in detail, including how to connect the individual certificates.

The formula F_{DAG} is constructed by the SMT parser from an input SMT2 file, or directly by the model checker. The instrumentation added to an SMT solver produces the *certificates* and a *DAG file*, a serialization of F_{DAG}, that in the case of unsatisfiability can be checked by the *certificate checker*, using different *checker modules*, and a *DAG checker*, that can be used to ensure the correctness of the construction of F_{DAG} from F with a straight-forward traversal. Figure 5.1 shows the components of the proposed system.

The core of the system is the SAT solving certificates, where clauses derived from both CNF conversion and theory solving are provided to the DRAT proof checker. It thus suffices to produce and check certificates for said clauses. In the following certificate production and checking for CNF conversion and the theories of linear arithmetics, for reals and integers, and uninterpreted functions with equality are discussed.

Input. The original formula F is transformed to its simplified version F_{DAG}. To validate the conversion, it suffices to traverse the representation of F and compare it to F_{DAG}.

CNF conversion. Certifying the CNF conversion consists of reproducing the standard way in which modern SMT solvers perform CNF conversion by applying the Tseitin transformation and the De Morgan rules. The certificate consists of a sequence of rule applications that maps the nodes N_B of F_{DAG} having the Boolean return sort to clauses in F_{CNF}, together with a bijection L between N_B and a set of literals. The checker verifies that L is a bijection, traverses F_{DAG} and, at each node in N_B, applies the appropriate rule based on the node's symbol's name and verifies that the correct set of clauses appears in F_{CNF}. This is done by applying L^{-1} to each literal in the candidate clauses and checking that they match correctly the node and its children. Identifiers can be used to point to clauses in the certificate to avoid the worst-case quadratic blow-up in the size of the CNF formula in cases where many nodes produce a shared clause.

SAT solving. To validate the computations done by the back-end SAT solver the DRAT proof format is used, which is based on the concept of clause redundancy. Having a CNF formula as input, a DRAT proof sequentially adds and deletes clauses from the formula while preserving unsatisfiability, with the last addition of a valid proof being that of the empty clause, interpreted as *false*.

Linear real arithmetic solving. To validate the theory clauses in linear real arithmetic contained in the CNF formula the conflicts that originated them are recreated. To do so, the conflict's explanations, i.e., the conjunction of literals that led to the conflict, are logged in the linear real arithmetic certificate, with the literals representing inequalities derived from the original formula F. Using the Farkas' lemma, a linear combination of the inequalities must lead to an inconsistency of form $1 \leq 0$ in a correct witness. The corresponding conjunction of inequalities is then negated and matched to the theory clause.

Linear integer arithmetic solving. The theory clauses in linear integer arithmetic can be validated by two different methods. The first method involves recreating the conflicts that happen in the real domain, using the same approach as done for real arithmetic, since an unsatisfiable result in the real domain implies the same result in the integer domain. The second method validates the tauto-

logical theory clauses of the form of integer bounds, given when a non-integer assignment is found, in order to restrict the search space. Since tautological clauses do not interfere with the satisfiability of the formula, we only check if they are well-formed bounds of the form $x \leq n \vee x \geq n+1$, where $n \in \mathbb{Z}$ and x is a variable in F_{DAG}.

Uninterpreted functions solving. The clauses of the theory of uninterpreted functions have a specific form, which corresponds to a conflict of a single disequality $t_1 \neq t_2$ and a set of equalities P from which an equality $t_1 = t_2$ can be derived. Moreover, this equality can be derived using the basic properties of logical equality: *symmetry*, *transitivity* and *congruence*. The uninterpreted functions certificate records the applications of the transitivity and the congruence rule in the derivation of the equality $t_1 = t_2$. It is possible to record the whole derivation in the run of the *Explain* procedure of the state-of-the-art congruence-closure algorithm of Nieuwenhuis and Oliveras [2005] with minimal changes. To validate the certificate it is sufficient to (recursively) check that the equality $t_1 = t_2$ is *well-derived*, where an equality is well-derived if it or its symmetrical counterpart (i) is an element of P, or (ii) has been derived using congruence or transitivity from well-derived equalities.

5.3 Implementation

The certificate production and the certificate checking parts of the approach were both implemented, as per Figure 5.1. The instrumentation was done on the freely available, MIT licensed, SMT solver OpenSMT [Bruttomesso et al., 2010]. The OpenSMT solver won the QF_LRA track of the 2020 edition of the annual SMT competition and is also competitive in the instances from the logics of QF_LIA and QF_UF. The instrumentation size, as reported by GIT-DIFF, is 1795 lines, excluding regression and unit tests. The certificate checking was implemented in the automated Theory-Specific Witness Checker, TSWC for short; both tools are available online[2].

Certificate production. The instrumented version of OpenSMT, which is called OpenSMT-C, accepts instances in the SMT-LIB standard [Barrett et al., 2021] and produces correctness certificates for each checker module, i.e., CNF conversion, SAT solving, and theory solving for QF_LRA, QF_LIA, and QF_UF, that can then be forwarded to a checker. A key point to avoid the introduction of a memory overhead due to certificate production is that all information relevant to the certificate is written to the disk immediately, instead of being stored in

[2]Available at `http://verify.inf.usi.ch/certificate-producing-opensmt2`.

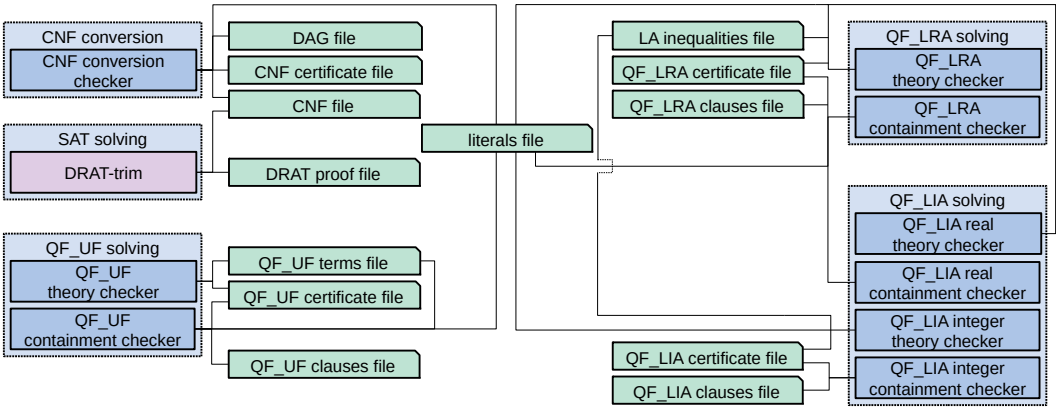


Figure 5.2. TSWC architecture. The nine components are represented as solid rectangles, and are grouped by checker modules they belong to; the components in blue were developed by us, and the one in purple is off-the-shelf. All the files used by TSWC are also shown, in green, with the lines indicating which files are used by each component.

an internal data structure of the solver. In the current implementation certain simple rewriting steps that OpenSMT performs while transforming F to F_{DAG} are omitted. These steps are thus currently left for the DAG checker. Great care was taken so that all the simplifications that are not validated can be performed by the DAG checker in the sense that checking them does not require knowledge of the SMT solver's data structures.

Certificate checking. The TSWC tool consists of nine independent components, one of them being the DRAT-trim proof checker [Wetzler et al., 2014] for certifying the results of the SAT solver, and the other eight being Python scripts with less than 300 lines of code each. It is believed that the compact and modular checker design makes the code base easier to inspect both manually and, in the future, in an automated fashion. An overview of TSWC can be seen in Figure 5.2. Starting with the CNF conversion, its checker receives the DAG, the CNF conversion certificate, and the CNF formula used by DRAT-trim, and it applies the rules listed by the certificate in order to validate the clauses on the CNF formula that are derived from the DAG. The remaining clauses in the CNF formula are theory clauses, with the purpose of the checkers for each theory being to validate them. Each theory has both a theory checker, that validates the theory certificate, and a containment checker, that checks if all theory clauses added to the CNF formula are certified; for QF_LIA both the checkers for real and integer arithmetics are used, since it has solving procedures from both domains. With the clauses derived from both the CNF conversion and the theory solving vali-

Table 5.1. Proof production comparison. For each theory, the number of instances classified as unsatisfiable, timeouts, memouts, and errors given by the solvers is reported; an error refers to an exception being thrown during execution. Each cell contains the result for standard mode on the left and for proof producing mode on the right, with the exception of those for % change, which report the variation on the number of instances classified as unsatisfiable.

		UNSAT	% change	Timeout	Memout	Error
QF_LRA (1648 instances)	OpenSMT-C	**636/633**	**99.5%**	**151/156**	0/0	0/0
	CVC5	583/570	97.7%	274/289	0/0	0/0
	veriT	580/561	96.7%	249/268	0/0	10/10
	Z3	570/558	97.8%	253/264	0/0	0/0
QF_LIA (6947 instances)	OpenSMT-C	2023/1519	75.0%	3341/3892	0/0	0/0
	CVC5	1398/928	66.3%	3147/3620	0/0	0/0
	veriT	1002/953	95.1%	4198/4247	0/0	1/1
	Z3	**2238/2340**	**104.5%**	**1786/1685**	1/1	0/0
QF_UF (7457 instances)	OpenSMT-C	4326/**4311**	**99.6%**	24/**39**	0/0	0/0
	CVC5	4321/4220	97.6%	34/135	0/0	0/0
	veriT	**4347**/4176	96.0%	**3**/166	0/0	0/8
	Z3	4341/4236	97.5%	9/114	0/0	0/0

dated, DRAT-trim can then ensure unsatisfiability at the SAT level. All auxiliary data is stored in the *literals*, *inequalities*, and *terms files*. The literals file contains a mapping between literals and nodes of F_{DAG}, the inequalities file contains the linear arithmetic literals, and the terms file contains the uninterpreted functions and their arguments.

5.4 Evaluation

In the evaluation the non-incremental benchmark sets of each theory supported by OpenSMT-C available in the SMT-LIB benchmark repository[3] were used. For proof production OpenSMT-C was compared against three proof producing solvers, namely CVC4 1.8, veriT 09a24ff-rmx, and Z3 4.8.9. The number of proofs produced and their sizes were measured, as well as the proof production time and overhead, in terms of runtime and memory use. For proof checking TSWC was compared against the LFSC checker, which can automatically check CVC4's proofs, and was, until recently, the only tool comparable to TSWC[4], in terms of

[3] See `http://smtlib.cs.uiowa.edu/benchmarks.shtml`.

[4] Recent developments are discussed in Section 6.2.

Table 5.2. Proof production comparison. For each theory, the solvers' average runtime, in seconds, and memory use and proof size, in kilobytes, is reported. Each cell contains the result for standard mode on the left and for proof producing mode on the right, with the exception of those for average proof size, whose results are only relevant in proof producing mode.

		Avg. runtime	Avg. mem. use	Avg. proof size
QF_LRA (1648 instances)	OpenSMT-C	**3.91**/**4.31**	30532/**26603**	3956
	CVC5	5.73/6.61	35877/76953	13684
	veriT	4.12/5.22	**19227**/44358	69438
	Z3	5.17/5.30	36472/76929	**3546**
QF_LIA (6947 instances)	OpenSMT-C	10.07/9.99	113622/51443	**13018**
	CVC5	6.30/**1.30**	63127/37728	18883
	veriT	**1.82**/4.05	**12028**/**20277**	165914
	Z3	5.17/6.62	58325/276318	24876
QF_UF (7457 instances)	OpenSMT-C	0.95/1.14	9416/**9563**	**6730**
	CVC5	0.39/1.83	17517/28316	6854
	veriT	**0.10**/**0.79**	**5931**/23940	20102
	Z3	0.22/1.26	15474/49355	12601

runtime and memory use. The experiments were done with a 60 seconds timeout and a 10 gigabytes memory limit; all results are available online[5].

Proof production. All solvers were ran in both standard and proof producing modes. The results are compiled in tables 5.1 and 5.2. For the number of proofs produced, OpenSMT-C had the best result for QF_LRA and QF_UF, also having the smallest overhead in number of proofs, while Z3 had by far the best results for QF_LIA. Regarding runtime, OpenSMT-C and veriT had the best performances for QF_LRA and QF_UF, respectively, while for QF_LIA CVC4 had the shortest runtime in proof producing mode. CVC4's average runtime for QF_LIA has, however, to be taken with a grain of salt, since its apparent increase in performance when proof production is enabled may be due to the sharp increase in the number of timeouts, which are not part of the computed average. When comparing memory use, OpenSMT-C had the best results in proof producing mode for QF_LRA and QF_UF, while veriT had the smallest use for QF_LIA. In terms of proofs sizes, OpenSMT-C had the best results for QF_LIA and QF_UF, while also having a competitive result for QF_LRA, with an average proof size close to that of Z3, which had the best result. A runtime, memory use, and proof size

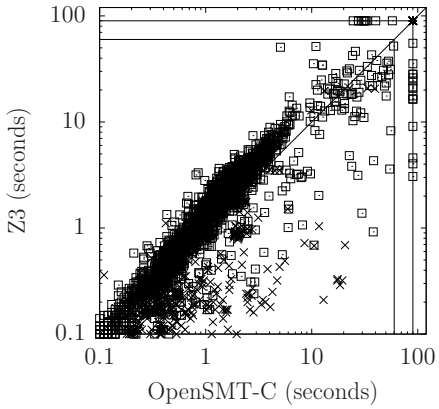
[5]Available at `https://scm.ti-edu.ch/repogit/verify-witness-evaluation.git`.

comparison of the two solvers with most proofs produced, for each theory, can be seen in figures 5.3, 5.4, and 5.5; all pairwise comparisons are in proof producing mode and are available online[5].

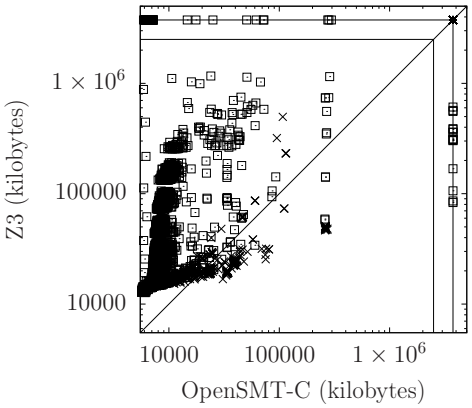
Proof checking. For all proofs produced by OpenSMT-C and CVC5 their respective checkers were ran, with the results being compiled in Table 5.3; no proof was rejected by either tool, nor did any memouts were registered. We can see that TSWC was able to verify more proofs for every theory. For runtime and memory use, LFSC had better results for two of the three theories, but this can be mainly attributed to the high number of errors it had, which are not part of the computed average.

5.5 Conclusions and Future Work

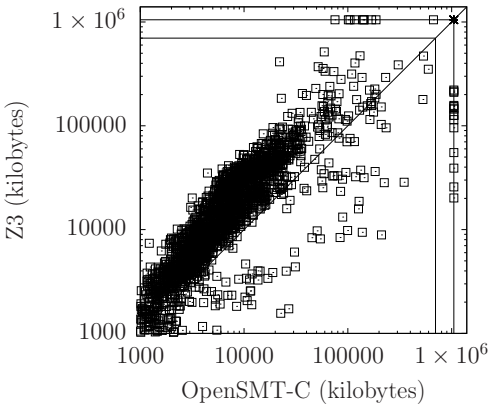
In this chapter it was showed that when a SMT solver claims that a query on an instance is unsatisfiable, the correctness of that claim can be checked using the data structures that the core solving algorithms of SMT are already maintaining, by connecting them to their underlying mathematical and logical foundations. In addition, it was also showed that it is easy to write checkers for these certificates, and producing them has a comparable or smaller overhead when contrasted with more traditional proofs.



(a) QF_UF runtime comparison.

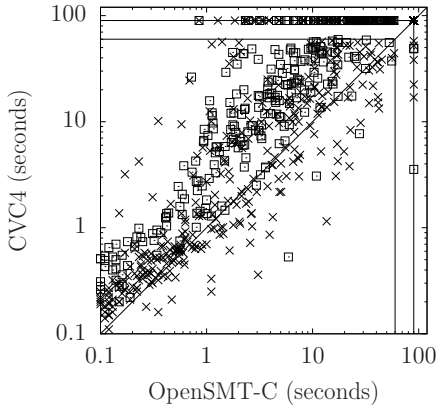


(b) QF_UF mem. use comparison.

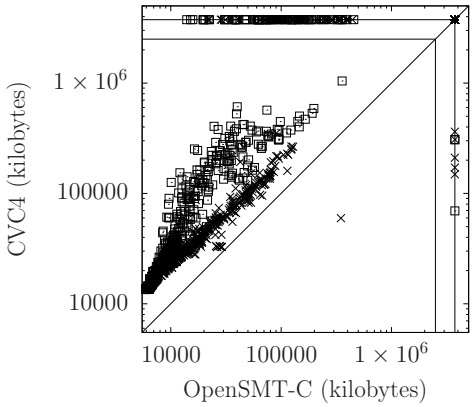


(c) QF_UF proof size comparison.

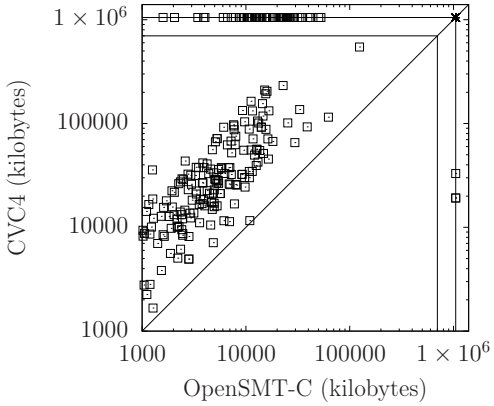
Figure 5.3. QF_UF results; squares and crosses stand for UNSAT and SAT, the top and right lines represent value limit, timeout, and memout.



(a) QF_LRA runtime comparison.

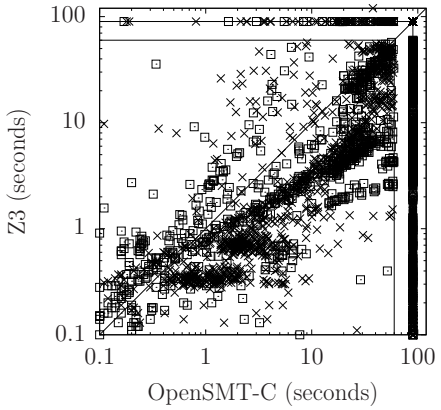


(b) QF_LRA mem. use comparison.

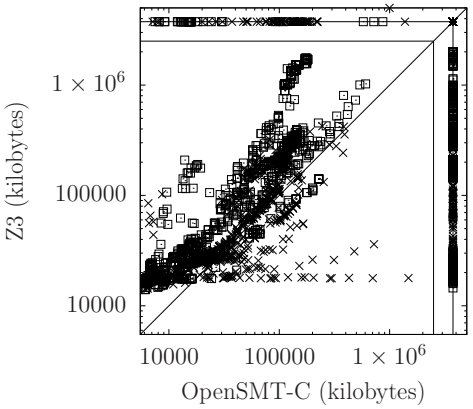


(c) QF_LRA proof size comparison.

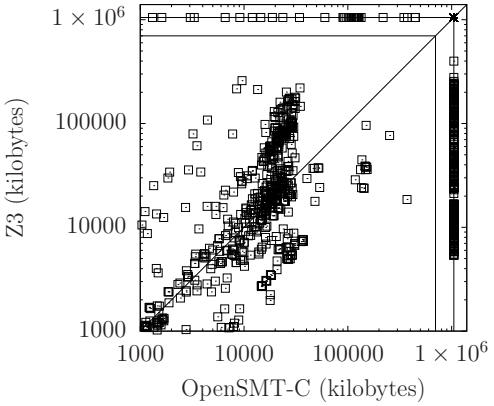
Figure 5.4. QF_LRA results; squares and crosses stand for UNSAT and SAT, the top and right lines represent value limit, timeout, and memout.



(a) QF_LIA runtime comparison.



(b) QF_LIA mem. use comparison.



(c) QF_LIA proof size comparison.

Figure 5.5. QF_LIA results; squares and crosses stand for UNSAT and SAT, the top and right lines represent value limit, timeout, and memout.

Table 5.3. Proof checking comparison. For each theory, the number of verified proofs, timeouts, and errors given by the checkers is reported, as well as the checkers' average runtime, in seconds, and memory use, in kilobytes; an error refers to an exception being thrown during execution. The instances used are those for which both OpenSMT-C and CVC4 produced a proof.

		Verified	Timeout	Error	Avg. runtime	Avg. mem. use
QF_LRA	TSWC	**564**	**3**	0	3.15	**67372**
(567 instances)	LFSC	471	8	88	**2.85**	102861
QF_LIA	TSWC	**903**	10	0	1.37	65039
(913 instances)	LFSC	128	**0**	785	**0.12**	**26609**
QF_UF	TSWC	**4217**	**1**	0	**1.44**	69507
(4218 instances)	LFSC	4157	50	11	4.01	**28454**

Directions for future work include (i) the extension of the approach to other SMT theories, (ii) the integration of proof producing solvers into SMT-based tooling, and (iii) the creation of a standard format for SMT unsatisfiability proofs. For the first direction it is believed that more applied theories, like the theory of arrays, are a suitable next step, with another possibility being the combination of already supported theories. For the second direction possibilities include integration into model checkers and CHC solvers. For the third direction a community effort is required for the establishment of a standard, which needs to both be expressive and lead to efficient proof production and checking.

Chapter 6

CHC Model Validation with Proof Guarantees

Different fragments of FOL are suitable to aid in specific verification tasks, with one fragment of particular practical interest being CHC [Gurfinkel and Bjørner, 2019], which has been used to aid in reasoning about the behaviour of procedural [Bjørner et al., 2015] and functional [Grebenshchikov et al., 2012] programs, as well as concurrent systems [Hojjat et al., 2014] and smart contracts [Marescotti et al., 2020]. CHC solvers, e.g., ELDARICA [Hojjat and Rümmer, 2018], GOLEM [Blicha et al., 2023], and SPACER [Komuravelli et al., 2016], serve, for instance, as the back-end reasoning engines of verification tools targeting programs written in C/C++ [Gurfinkel et al., 2015], Java [Kahsai et al., 2016], Rust [Matsushita et al., 2021], and Solidity [Alt et al., 2022], as well as Android applications [Calzavara et al., 2016].

Despite their extensive usage in verification, logic solvers are themselves not immune to bugs. When it comes to CHC solvers, the annual CHC competition, CHC-COMP, encountered similar issues to its SAT and SMT counterparts, with competing solvers disagreeing on certain benchmarks, and its organizers having the validation of results as a goal [de Angelis and Govind V. K., 2022]. For CHC, a witness for an UNSAT result, called an UNSAT proof, should contain an explanation of how *false* can be derived, while a witness for a SAT result, called a SAT model, should contain interpretations for all the predicates in such a way that all clauses evaluate to *true*, entailing that *false* cannot be derived; from now on UNSAT proofs and SAT models will be referred to simply as proofs and models.

This chapter focuses on the validation of CHC models. While the production of witnesses, be they models or proofs, is a common feature of modern CHC solvers, efforts in witnesses validation are limited at present. The validation of

models is done via SMT queries, and is currently supported only by an ad hoc validator tied to the SMT solver Z3[1]. A novel proof-backed validation approach for CHC models is proposed, that, in addition to providing additional correctness guarantees, is solver independent. The ATHENA framework was developed to implement the approach and a large scale evaluation involving twelve tools in total was carried out, in order to showcase its practicality and benefits.

6.1 Overview

Since CHC model validation is underpinned by SMT solving, the same concern regarding the correctness of CHC solvers' results is put on the validation itself, i.e., on the correctness of SMT solvers' results. To address this, a two-layered validation approach is proposed to provide additional guarantees about the results obtained, illustrated in Figure 6.1. The first layer, consisting of the SMT queries responsible for model validation, is enhanced by a second layer, consisting of the production and checking of SMT proofs, with the result obtained being forwarded to the user or tool interacting with the CHC solver. The approach is generic w.r.t. FOL theories and solvers, and is also very modular, enabling different SMT solvers to be used in the validation, further increasing assurances.

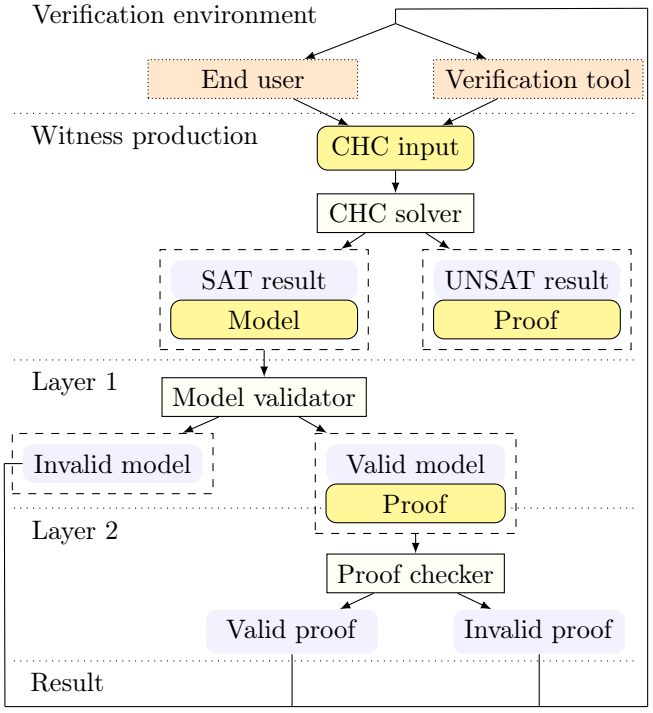


Figure 6.1. The two-layered validation approach for CHC models. Although capable of being produced, CHC proofs cannot be checked currently.

To asses the viability of practical model validation the modular evaluation framework ATHENA was developed, capable of catering to different combinations of state-of-the-art CHC and SMT solvers,

[1]See `https://github.com/chc-comp/chc-tools/blob/master/chctools/chcmodel.py`.

Table 6.1. Brief descriptions of the bugs found during the evaluation.

		Bug Effect
CHC solvers	ELDARICA	Invalid model produced[6]
	SPACER	Invalid model produced[8]
	GOLEM	Syntactically malformed model produced[5]
	GOLEM	Crash during model production[4]
SMT solvers	CVC5	Invalid proof produced[11]
	CVC5	Crash during proof production[10]
	OPENSMT	Crash during sort inference[9]
Proof checkers	CARCARA	Parsing error due to unknown attribute[12]
	LFSC checker	Crash during type inference[13]

and a large scale evaluation was conducted. Concretely, the framework was used to validate the models produced by three CHC solvers, ELDARICA [Hojjat and Rümmer, 2018], GOLEM [Blicha et al., 2023], and SPACER [Komuravelli et al., 2016], with each model produced being separately validated, in Layer 1, by five proof producing SMT solvers, CVC5 [Barbosa et al., 2022a], OPENSMT [Bruttomesso et al., 2010], SMTINTERPOL [Christ et al., 2012], VERIT [Bouton et al., 2009], and Z3 [de Moura and Bjørner, 2008b]. In addition, all the proofs produced in the proof formats currently supported by automated proof checkers were checked, in Layer 2, by using the checkers CARCARA [Andreotti et al., 2023], LFSC checker [Stump et al., 2013], SMTINTERPOL checker [Hoenicke and Schindler, 2022], and TSWC [Otoni et al., 2021].

To have a focused evaluation the experiments were conducted on benchmarks from one specific FOL theory, the LIA theory. All 955 LIA benchmarks from CHC-COMP 2022 were used in the evaluation, 499 containing only linear Horn clauses, i.e., implications with a single uninterpreted predicate in the implicant, and 456 containing nonlinear Horn clauses, i.e., implications with multiple uninterpreted predicates in the implicant. The benchmarks led to 91626 SMT instances and 385303 SMT proofs being produced during the validation process.

Three observations can be made from the results obtained. First, the proof-backed model validation approach proposed is viable in practice, with the majority of the models being validated with available tooling. This means that any CHC-based tool, e.g., the SEAHORN [Gurfinkel et al., 2015] and SOLCMC [Alt et al., 2022] model checkers, can in principle benefit from the guarantees provided by model validation. Second, model and proof sizes, which were in the

order of hundreds of megabytes in the experiments and can potentially require gigabytes of storage, are a concern and a potential limitation to the practical use of validation. Producing compact models and proofs is thus an important goal, with compression, recently investigated in the context of unsatisfiability proofs for SAT solvers by Reeves et al. [2023], being a potential complementary goal. Lastly, model validation provides a useful way to generate new and interesting SMT instances. The evaluation uncovered bugs not only in the selected CHC solvers, which are the main focus, but also in two SMT solvers and two proof checkers for SMT proofs. The bugs found, listed in Table 6.1, range from parsing errors to invalid models being produced. They were all acknowledged by the developers and are detailed in Section 6.5. In addition to aiding in tool development, these bugs confirm the need for additional guarantees to be provided to modern verification tooling.

6.2 Related Witness Validation Approaches

As logic solvers became more powerful they were quickly adopted as the back-end reasoning engines of many verification tools. The need to validate the answers from these solvers arose soon after, with the complexity of the validation increasing hand-in-hand with the expressiveness of the underlying formalism.

In line with its relative simplicity, witness validation in the context of Boolean satisfiability was the first to be investigated. Validating a satisfying model is an easy task: one simply substitutes the variables of the formula with their values from the model and checks if the resulting Boolean expression over constants *true* and *false* simplifies to *true*. The validation of unsatisfiability proofs, however, is far from trivial, even in such a restricted domain. Many proof formats have been proposed, offering different trade-offs between proof compactness and checking efficiency. Initial formats were based on resolution [Sinz and Biere, 2006] and clausal proofs [Heule et al., 2013a], with resolution asymmetric tautology (RAT) [Heule et al., 2013b] being a base for many recent developments, e.g., deletion RAT (DRAT) [Wetzler et al., 2014], linear RAT (LRAT) [Cruz-Filipe et al., 2017], and flexible RAT (FRAT) [Baek et al., 2021]. The production of proofs in the DRAT format has been a requirement in the SAT competition since its 2014 edition, with DRAT-TRIM [Wetzler et al., 2014] being the standard proof checker for proofs following this format.

In regards to satisfiability modulo theories, witness validation is complicated by the presence of theories and quantifiers. No standard way of representing SMT models currently exists, with a consistent push by the SMT competition or-

ganizers having been made in recent years for the adoption of a unified format in line with the SMT-LIB standard [Barrett et al., 2021]. A separate, experimental, model validation track has been established and has seen a steady increase in the SMT-LIB logics supported, with PySMT [Gario and Micheli, 2015] and Dolmen [Bury, 2021] used as validating tools. Despite recent advances, model validation is still restricted to quantifier-free formulas. For unsatisfiability proofs, different formats, often attached to a specific solver, have been proposed. The Alethe format [Schurr et al., 2021] was initially supported by veriT, but has since also being integrated into CVC5's proof production. CVC5 also caters for proofs based on the logical framework with side conditions (LFSC) [Stump et al., 2013], with LFSC support preceding Alethe's integration and dating back to the CVC3 version of the tool. SMTInterpol [Hoenicke and Schindler, 2022], OpenSMT [Otoni et al., 2021], and Z3 [de Moura and Bjørner, 2008a] also support their own, unnamed, proof formats. Each format has one or more associate tools that can consume the proofs produced, with said tools being either interactive or automated. In the interactive side, proof assistants discharge some verification conditions to external logic solvers as a way to increase their level of automation, with the proofs produced providing new theorems to be checked by the proof assistant's internal engine, as it has been done with Coq [Armand et al., 2011; Ekici et al., 2017] and Isabelle/HOL [Fontaine et al., 2006; Böhme and Weber, 2010; Blanchette et al., 2016; Barbosa et al., 2020]. When it comes to automated checkers, their goal is mainly to serve as independent lightweight validators, with potential to be integrated into tools such as model checkers. Automated checkers are available for a variety of formats [Stump et al., 2013; Wetzler et al., 2014; Otoni et al., 2021; Hoenicke and Schindler, 2022; Andreotti et al., 2023]. As of the time of writing, no proof format is enforced by the SMT competition, with an experimental track being available as a way to showcase the strengths of existing formats.

In addition to logic solving, witness validation is also pursued in other contexts. A good example of this is the use of validation in the annual competition on software verification [Beyer, 2023]. Software verification witnesses are different from those used by logic solvers, being categorizes as either correctness or violation witnesses, with their own formats[2] and limitations [Beyer and Strejček, 2022]. The tool that maybe best illustrates usage of witness is Korn [Ernst, 2023], a participant in the software verification competition that relies on Horn solvers as its back-end and produces witnesses for its reasoning about C programs' properties from the witnesses produced by the underlying solvers.

[2] See `https://gitlab.com/sosy-lab/benchmarking/sv-witnesses`.

6.3 Validation of CHC Models

A CHC solver is a complex piece of software, often implementing sophisticated algorithms relying on decision and interpolation procedures, which allows for subtle bugs to occur and lead to incorrect answers. In addition to providing much needed stronger guarantees in regards to SAT or UNSAT results, model validation also ensures the correctness of the models themselves, which are commonly relied upon by verification tools to, for instance, establish inductive invariants of programs. Model validation is therefore critical for assurance not only of solvers' results, but also of all structures derived from models presented to end users.

A two-layered validation approach for CHC models was proposed, detailed in Figure 6.2; since the focus is on models, the illustration assumes the benchmark is satisfiable. The first of the two layers in the approach handles model validation via SMT solving. Following the CHC definition laid out in Section 2.5, model validation can be done via a number of SMT queries which is linear w.r.t. to the amount of Horn clauses present in the input. Each such query checks if a specific Horn clause is logically valid in the theory T after its uninterpreted predicates are substituted by their interpretations given by the model. This is done by checking if the negation of the Horn clause, augmented with the interpretations, is satisfiable, i.e., if a satisfying assignment for $\phi \wedge \mathcal{P}_1 \wedge \ldots \wedge \mathcal{P}_n \wedge \neg H$ exists. This check is well suited for SMT solving, with a valid model leading to all queries being unsatisfiable. An important note is that, depending on the theory T, the query checking might be intractable for ex-

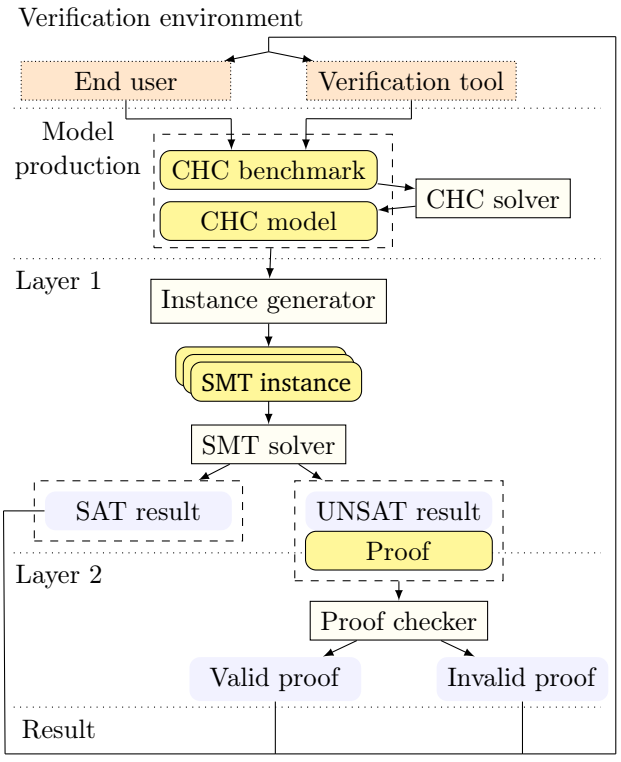


Figure 6.2. Breakdown of the two-layered validation approach for CHC models. A valid model will have all the SMT instances generated from it yield an UNSAT result backed by a valid proof.

isting SMT solvers, and in some cases even undecidable.

As an example, consider the following CHC system, consisting of three Horn clauses and a single uninterpreted predicate *Inv*:

$$\begin{aligned} Inv(x) &\leftarrow x \leq 0 \\ Inv(x') &\leftarrow Inv(x) \wedge x < 5 \wedge x' = x + 1 \\ false &\leftarrow Inv(x) \wedge \neg(x < 10) \end{aligned}$$

This system is satisfiable with a potential model being one that contains the interpretation $Inv(x) \equiv x \leq 5$. To validate this model we need to establish that the following three formulas are logically valid in the LIA theory:

$$\begin{aligned} x &\leq 5 \leftarrow x \leq 0 \\ x' &\leq 5 \leftarrow x \leq 5 \wedge x < 5 \wedge x' = x + 1 \\ false &\leftarrow x \leq 5 \wedge \neg(x < 10) \end{aligned}$$

The validation can be done by showing that the three formulas below are unsatisfiable, which can be trivially seen for this small example:

$$\begin{aligned} \neg(x &\leq 5) \wedge x \leq 0 \\ \neg(x' &\leq 5) \wedge x \leq 5 \wedge x < 5 \wedge x' = x + 1 \\ \neg false &\wedge x \leq 5 \wedge \neg(x < 10) \end{aligned}$$

While it can be easy to validate models such as the one above, this is far from the case when dealing with real world examples. As a consequence, SMT solvers are, like their CHC counterparts, very complex tools that are susceptible to bugs. The second layer in the proposed approach tackles this issue via the validation of SMT solvers' results. A number of SMT solvers produce unsatisfiability proofs that can be independently checked. These proofs provide much needed guarantees regarding unsatiafiability results, which are at the core of the validation done in Layer 1. By relying on the currently untapped power of SMT proofs additional correctness guarantees can be provided for CHC model validation.

The approach is theory independent and can be applied to any CHC, with the only requirement being that a proof producing SMT solver and a proof checker are available for the theory in question. In addition to validating direct end user usage of CHC solvers, the approach can also be embedded into CHC-based verification tools, enhancing their own guarantees.

6.4 Implementation

To enable the practical use of the approach, with the immediate goal of ascertaining the capabilities of state-of-the-art CHC and SMT solvers, we developed the modulAr consTrained Horn clauses modEl validatioN frAmework, ATHENA for short. The framework is capable of validating CHC models via SMT solving while using different solver combinations. ATHENA also handles the production and checking of SMT proofs, for the SMT solvers with proof production capabilities. In addition, metrics such as model and proof sizes can be gathered and analysed. The framework consists of 2535 lines of shell and Python code in total, is fully automated, and uses GNU PARALLEL [Tange, 2011] to achieve a large degree of parallelisation in order to better tackle the high computation cost. ATHENA is open-source[3], enabling third-parties to make full use of it, with one of the goals being to provide the groundwork for model validation at CHC-COMP.

6.5 Evaluation

We first describe the benchmarks and tools used, in Section 6.5.1, and then discuss the results obtained related to CHC model validation, in Section 6.5.2, and SMT proof checking, in Section 6.5.3. A machine with 64 AMD EPYC 7452 processors and 256 GB of memory was used for the evaluation. All individual tool executions had a timeout of 60 seconds and a memory limit of 5 gigabytes.

6.5.1 Benchmarks and Tools

The benchmarks of the two LIA tracks of CHC-COMP 2022 [de Angelis and Govind V. K., 2022] were used, the LIA-lin track, consisting of benchmarks containing only linear Horn clauses, and the LIA-nonlin track, consisting of benchmarks containing nonlinear Horn clauses. It was decided to use LIA benchmarks for two reasons: first, the LIA tracks are the most traditional in CHC-COMP, being present in every edition of the competition and having the most competing solvers, and second, the LIA theory is covered by all proof producing SMT solvers available, even if for some only in its quantifier-free fragment.

For model production the current three best performing CHC solvers in the LIA tracks were chosen for comparison, which are, in alphabetical order, ELDARICA [Hojjat and Rümmer, 2018], GOLEM [Blicha et al., 2023], and SPACER [Komuravelli et al., 2016]. For model validation all SMT solvers that competed in

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

Table 6.2. Results for solving the CHC benchmarks of the two LIA tracks.

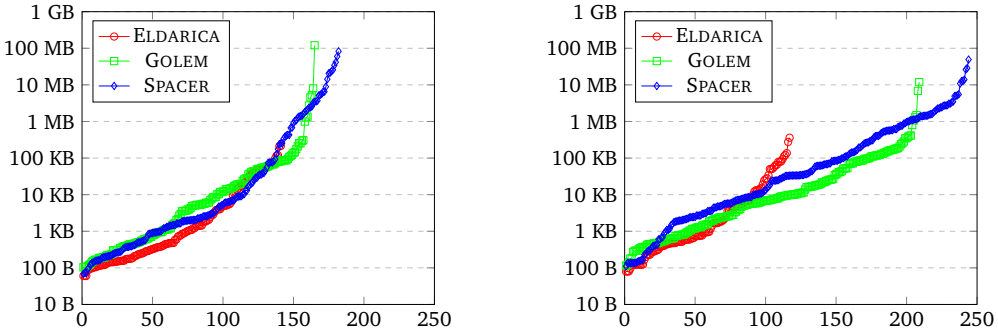
		SAT	UNSAT	Unknown	Timeout	Memout	Error
LIA-lin (499 benchmarks)	ELDARICA	141	51	0	307	0	0
	GOLEM	165	80	0	254	0	0
	SPACER	182	89	0	212	16	0
LIA-nonlin (456 benchmarks)	ELDARICA	117	56	0	283	0	0
	GOLEM	209	118	0	129	0	0
	SPACER	244	130	1	74	7	0

the proof exhibition track of the 2022 SMT competition were used, which are, in alphabetical order, CVC5 [Barbosa et al., 2022a], OPENSMT [Bruttomesso et al., 2010], SMTINTERPOL [Christ et al., 2012], and VERIT [Bouton et al., 2009], as well as Z3 [de Moura and Bjørner, 2008b], which can produce proofs but did not compete in the track. To check the SMT proofs the fully automated checkers CARCARA [Andreotti et al., 2023], for ALETHE proofs produced by CVC5 and VERIT, LFSC checker [Stump et al., 2013], for LFSC proofs produced by CVC5, SMTINTERPOL checker [Hoenicke and Schindler, 2022], for proofs produced by SMTINTERPOL, and TSWC [Otoni et al., 2021], for proofs produced by OPENSMT, were used; to the best of our knowledge there is currently no independent automated checker for proofs produced by Z3.

6.5.2 Model Validation Results

To produce the CHC models the selected CHC solvers were executed with all benchmarks; the results are summarised in Table 6.2. All tools were executed with their default engine configurations. The performance of the tools is in line with the CHC-COMP results, with SPACER solving the most benchmarks, followed by GOLEM and ELDARICA. Only one execution, with SPACER, yielded an `unknown` result, meaning that the solver terminated within the allocated time frame but was not able to decide if the benchmark was satisfiable or not. A number of errors, i.e., tool crashes, were observed with GOLEM while testing the framework[4], as well as syntactically malformed models being produced by it[5], with the underlying causes of both issues being addressed before the full-scale evaluation. Regarding the models' sizes, ELDARICA's models tended to be the most compact, followed by GOLEM's, with SPACER producing most of the larger models, as can

[4]See `https://github.com/usi-verification-and-security/golem/issues/29`.
[5]See `https://github.com/usi-verification-and-security/golem/issues/27`.



(a) Models for LIA-lin. (b) Models for LIA-nonlin.

Figure 6.3. Sizes of the CHC models produced. The models are ordered according to their size, the x-axis indicates their position in the order and the y-axis indicates their size.

be seen in Figure 6.3; the single largest model is an outlier produced by GOLEM, with a size exceeding 100 MB. The last point of note is that all models produced by GOLEM are quantifier-free, while ELDARICA produced 1 quantified model, for 1 nonlinear benchmark, and SPACER produced 281 quantified models in total, 90 from linear benchmarks and 191 from nonlinear benchmarks.

To validate each model the SMT instances generated from it were executed with the selected SMT solvers; in this section all the reported SMT solvers' executions were done with proof production disabled. One SMT instance is generated for each Horn clause in the CHC benchmark for which the model was produced, thus many SMT instances, sometimes hundreds, can be generated for a single model. The SMT instances generated for the models produced by each CHC solver are considered, by track, as separate instance sets, thus there are three instance sets per track. The results for the LIA-lin and LIA-nonlin instance sets can be seen in tables 6.3 and 6.4; the number of SMT instances generated for each CHC solver is related to the amount of models it produced.

The validation results provide a useful insight into the quality of the models produced by each CHC solver. The models produced by GOLEM are the only ones for which no invalid result, i.e., a SAT output, was observed. Both ELDARICA and SPACER produced invalid models, although the latter to a significantly higher degree. ELDARICA's invalid models are due to the embedding of Boolean values into arithmetic operations[6], which leads to an error in most SMT solvers, but can be solved by Z3 via unit propagation[7]. SPACER's invalid models are due to problem-

[6]See `https://github.com/uuverifiers/eldarica/issues/51`.
[7]See `https://github.com/Z3Prover/z3/issues/6719`.

Table 6.3. Results for solving the SMT instances generated from the LIA-lin models. Due to the space limitation, unknown, timeout, memout, and error are shortened to UNK, TO, MO, and ERR. UNS stands for unsupported, meaning that the solver is not equipped to handle some features of the instance.

		SAT	UNSAT	UNK	TO	MO	ERR	UNS
	CVC5	0	5041	0	0	0	9	0
LIA-lin	OpenSMT	0	4970	0	0	0	80	0
Eldarica	SMTInterpol	0	5041	0	0	0	9	0
(5050 instances)	veriT	0	4986	0	0	0	9	55
	Z3	3	5047	0	0	0	0	0
	CVC5	0	5268	0	0	0	0	0
LIA-lin	OpenSMT	0	5268	0	0	0	0	0
Golem	SMTInterpol	0	5265	0	3	0	0	0
(5268 instances)	veriT	0	5216	0	0	0	0	52
	Z3	0	5268	0	0	0	0	0
	CVC5	695	15464	0	73	0	0	0
LIA-lin	OpenSMT	7	700	0	0	0	912	14613
Spacer	SMTInterpol	105	11909	28	4190	0	0	0
(16232 instances)	veriT	19	1543	0	0	0	0	14670
	Z3	690	15026	0	516	0	0	0

atic internal transformations[8]. Another aspect of model quality is the presence of quantifiers, which can make solving harder and is unsupported by both OpenSMT and veriT. Two last points of note are the high number of OpenSMT errors, i.e., crashes, when handling instances generated from Eldarica and Spacer models, which is due to a limitation in scoping in the presence of different sorts[9], and the small, but consistent, number of instances unsupported by veriT. After a manual inspection, it was discovered that the lack of support observed with veriT is due to the LIA tracks containing some benchmarks that, although semantically belonging to the LIA fragment of FOL, use operators reserved for the nonlinear integer arithmetic (NIA) logic of the SMT-LIB standard; the competition organizers were informed of this finding and stated that this will be addressed.

[8] See `https://github.com/Z3Prover/z3/issues/6716`.

[9] See `https://github.com/usi-verification-and-security/opensmt/issues/613`.

Table 6.4. Results for solving the SMT instances generated from the LIA-nonlin models. Due to the space limitation, unknown, timeout, memout, and error are shortened to UNK, TO, MO, and ERR. UNS stands for unsupported, meaning that the solver is not equipped to handle some features of the instance.

		SAT	UNSAT	UNK	TO	MO	ERR	UNS
LIA-nonlin Eldarica (6216 instances)	CVC5	0	6216	0	0	0	0	0
	OpenSMT	0	2493	0	0	0	3706	17
	SMTInterpol	0	6216	0	0	0	0	0
	veriT	0	6195	2	0	0	0	19
	Z3	0	6216	0	0	0	0	0
LIA-nonlin Golem (22458 instances)	CVC5	0	22458	0	0	0	0	0
	OpenSMT	0	22458	0	0	0	0	0
	SMTInterpol	0	22458	0	0	0	0	0
	veriT	0	22449	0	0	0	0	9
	Z3	0	22458	0	0	0	0	0
LIA-nonlin Spacer (36402 instances)	CVC5	147	36254	0	1	0	0	0
	OpenSMT	0	961	0	0	0	1326	34115
	SMTInterpol	97	34095	764	1446	0	0	0
	veriT	0	2286	0	0	0	0	34116
	Z3	147	36230	0	25	0	0	0

6.5.3 Proof Checking Results

To validate the UNSAT results given by the SMT solvers we rely on the proofs produced by them. For each SMT instance generated from a CHC model the selected SMT solvers were executed in proof production mode. The number of proofs produced by each SMT solver can be seen in Table 6.5. Since proof production adds an overhead to solver execution, the number of proofs produced is expected to be lower than the amount of UNSAT results reported in tables 6.3 and 6.4. Concretely, the combined percentage of proofs produced in relation to the previous UNSAT results, for the six instance sets, is: 95.59% for CVC5-Alethe, 99.27% for CVC5-LFSC, 100% for OpenSMT, 96.05% for SMTInterpol, 0% for veriT, and 99.76% for Z3. The reduction in performance is overall small, with OpenSMT showing no performance degradation and CVC5-LFSC and Z3 having less than 1% reduction. Two points of note are that Alethe proofs led to more than six times the overhead than LFSC proofs in CVC5, and that veriT was not able to produce any proofs. The reason for the behaviour observed

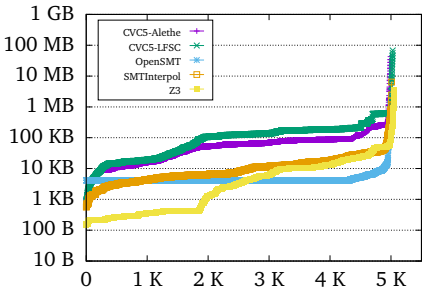
Table 6.5. Number of proofs produced by the selected SMT solvers; CVC5 has separate results for its two proof formats. Each column shows the amount of proofs produced from a given instance set, with the total amount of instances in each set shown below the CHC solver that produced the models for it.

	Proofs Produced					
	LIA-lin			LIA-nonlin		
	ELDARICA (5050)	GOLEM (5268)	SPACER (16232)	ELDARICA (6216)	GOLEM (22458)	SPACER (36402)
CVC5-ALETHE	4992	5169	12719	6116	22282	35419
CVC5-LFSC	5028	5226	14873	6216	22454	36234
OPENSMT	4970	5268	700	2493	22458	961
SMTINTERPOL	5010	5222	9548	6062	22299	33486
VERIT	0	0	0	0	0	0
Z3	5047	5268	14807	6216	22458	36230

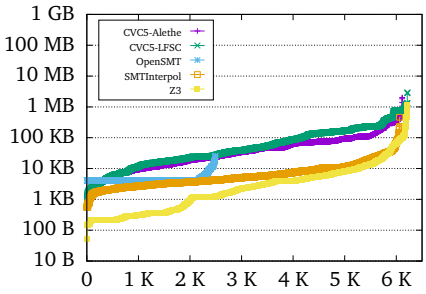
with VERIT is that the `define_fun` construct of the SMT-LIB standard, present in the models produced by all CHC solvers, is supported by VERIT in its default configuration, but not in its proof production mode. In addition, 117 new errors were observed with CVC5, which only happened in proof production mode, due to an unexpected free assumption leading to a fatal failure[10].

The proof formats used by each SMT solver can be quite different, not only in shape, but also in the amount of information stored, with the choice of finer or coarser proofs potentially having a significant effect on proof size. The sizes of all proofs produced in the evaluation can be seen in Figure 6.4. Overall, CVC5 produced the largest proofs, in both of its proof formats, in some cases with an order of magnitude difference with the proofs produced by the solver with the third largest proofs. The ranking between OPENSMT, SMTINTERPOL, and Z3 depends on which CHC solver's models the instances are generated from. A large number of Z3 proofs, all with a size of 50 B, consisted of `(proof (asserted false))`, showcasing how coarse proofs can be; although very compact, these extreme examples make checking essentially degenerate into solving the instance again. Regarding the CHC solvers themselves, ELDARICA's models led to the majority of the largest proofs for LIA-lin instances and SPACER' models led to the majority of the largest proofs for LIA-nonlin instances. The single largest proof produced, by CVC5-ALETHE from an instance generated from a SPACER LIA-nonlin model, had

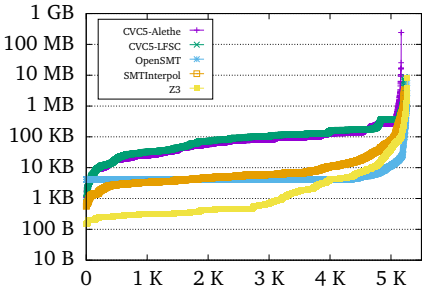
[10] See `https://github.com/cvc5/cvc5/issues/9770`.



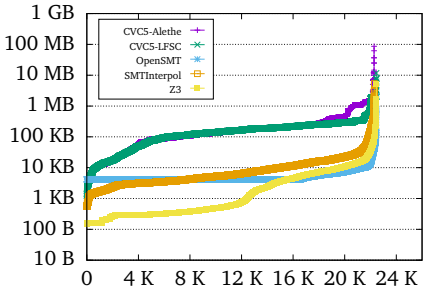
(a) Proofs for LIA-lin Eldarica.



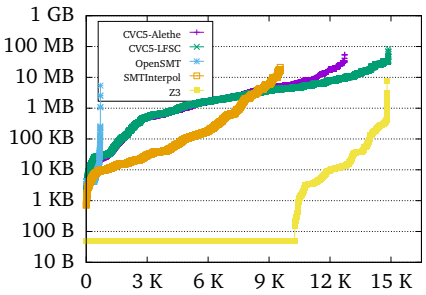
(b) Proofs for LIA-nonlin Eldarica.



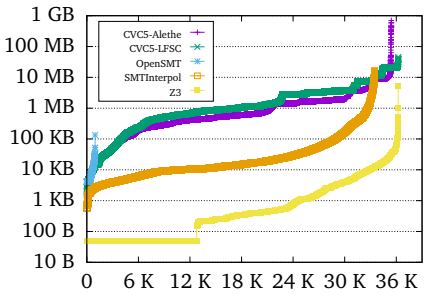
(c) Proofs for LIA-lin Golem.



(d) Proofs for LIA-nonlin Golem.



(e) Proofs for LIA-lin Spacer.



(f) Proofs for LIA-nonlin Spacer.

Figure 6.4. Sizes of the SMT proofs produced. The proofs are ordered according to their size, the x-axis indicates their position in the order and the y-axis indicates their size; the scale of the x-axis changes between the plots, due to the high variation on the number of proofs produced.

a size of 699 MB, which is a good illustration of the need of compact proofs.

To check the proofs the available automated checkers suitable for each proof format are used, namely CARCARA and TSWC for the proofs produced by CVC5-ALETHE and OPENSMT, and the LFSC and SMTINTERPOL checkers for the proofs produced by CVC5-LFSC and SMTINTERPOL. The results for the proofs produced for the LIA-lin and LIA-nonlin instance sets can be seen in tables 6.6 and 6.7.

Table 6.6. Results for checking the proofs produced by solving the SMT instances generated for the LIA-lin benchmarks; LFSC and SMTInterpol stand for their respective checkers. In addition to the raw number of proofs verified, the percentage relation to the total number of proofs is in parentheses.

		Valid	Invalid	Timeout	Memout	Error
LIA-lin ELDARICA	CARCARA	4992 (100%)	0	0	0	0
	LFSC	5026 (99.9%)	0	2	0	0
	SMTINTERPOL	5010 (100%)	0	0	0	0
	TSWC	4970 (100%)	0	0	0	0
LIA-lin GOLEM	CARCARA	5038 (97.4%)	131	0	0	0
	LFSC	5214 (99.7%)	0	7	3	2
	SMTINTERPOL	5222 (100%)	0	0	0	0
	TSWC	5268 (100%)	0	0	0	0
LIA-lin SPACER	CARCARA	1478 (11.6%)	109	0	0	11132
	LFSC	11295 (75.9%)	0	7	3570	1
	SMTINTERPOL	9542 (99.9%)	0	6	0	0
	TSWC	700 (100%)	0	0	0	0

Overall, the four checkers were able to validate most of the proofs produced, with the LFSC checker being the only tool to be significantly affected by the resource constraints, specifically the memory limit of 5 gigabytes. An important discovery is that CVC5-ALETHE produced 562 invalid proofs, due to incorrect proof steps[11]. While not implying that the UNSAT results the proofs are supposed to validate are incorrect, since the problem can be in the proof production itself, this is a serious issue. Still in regards to ALETHE proofs, CARCARA had 44282 errors when checking proofs produced for SMT instances generated from SPACER models, due to the presence of attribute annotations in models containing quantifiers[12]. Lastly, 3 errors were also observed with the LFSC checker, due to a type mismatch[13].

6.6 Conclusions and Future Work

In this chapter a novel two-layered approach for CHC model validation that relies on SMT proofs to provide additional correctness guarantees was presented. The

[11] See `https://github.com/cvc5/cvc5/issues/9760`.
[12] See `https://github.com/ufmg-smite/carcara/issues/12`.
[13] See `https://github.com/cvc5/LFSC/issues/87`.

Table 6.7. Results for checking the proofs produced by solving the SMT instances generated for the LIA-nonlin benchmarks; LFSC and SMTInterpol stand for their respective checkers. In addition to the raw number of proofs verified, the percentage relation to the total number of proofs is in parentheses.

		Valid	Invalid	Timeout	Memout	Error
LIA-nonlin Eldarica	Carcara	6115 (99.9%)	1	0	0	0
	LFSC	6216 (100%)	0	0	0	0
	SMTInterpol	6062 (100%)	0	0	0	0
	TSWC	2493 (100%)	0	0	0	0
LIA-nonlin Golem	Carcara	21999 (98.7%)	283	0	0	0
	LFSC	22453 (99.9%)	0	1	0	0
	SMTInterpol	22299 (100%)	0	0	0	0
	TSWC	22458 (100%)	0	0	0	0
LIA-nonlin Spacer	Carcara	2231 (6.2%)	38	0	0	33150
	LFSC	36222 (99.9%)	0	3	9	0
	SMTInterpol	33468 (99.9%)	0	18	0	0
	TSWC	961 (100%)	0	0	0	0

approach is supported by a modular evaluation framework, ATHENA, that enables models to be validated by many different SMT solvers and the SMT solving results to be validated by available proof checkers. A large scale evaluation was conducted using all LIA benchmarks from CHC-COMP 2022 to compare three CHC solvers, five SMT solvers, and four proof checkers. The results indicate that the approach is feasible in practice, with potential to benefit CHC-based verification tools, and also highlight model and proof sizes as a crucial practicality factor. A final important point is that many bugs were found in the tools compared, including invalid models being produced by two state-of-the-art CHC solvers, which confirms the need to provide modern verification tooling with additional correctness guarantees.

Directions for future work include (i) the evaluation of the approach with other FOL theories, (ii) the embedding of the approach into CHC-based verification tooling, and (iii) the designing of a complementary approach to validate CHC proofs. For the first direction, enhancements can be made to the framework's implementation to cater to theories other than LIA, with a point of interest being the checker support for SMT proofs not involving arithmetics. For the second direction, the use of proof-backed model validation in CHC-based model checkers is a direct application. For the third direction, one possibility is to use

the ALETHE format to represent and check CHC proofs, since it is rich enough to describe the necessary proof steps. An important unknown regarding potential ALETHE CHC proofs is the correct level of granularity, as it is unclear if coarse proofs can be efficiently checked, either by CARCARA or any future checker, or if additional burden needs to be put on the solvers to produce fine-grained proofs.

Chapter 7

Conclusions

This dissertation addressed the need for automated verification of blockchain technologies with correctness guarantees. The blockchain space was approached both at the platform and the application level and investigated the use of verification techniques based on first-order logic to tackle the growing need of correctness guarantees in this domain. To strengthen the guarantees provided, the use of correctness witnesses to validated the results of logic solvers dedicated to logic fragments of interest was also investigated.

In Chapter 3 a novel symbolic model checking approach for TLA$^+$ was proposed, with the goal of improving performance. The essence of the approach is the encoding of structural information of TLA$^+$ data structures into SMT, by usage of the theory of arrays, which leads to SMT formulas that are smaller and whose solving can be done more efficiently. This was implemented in the APALACHE model checker and allowed the automated reasoning of complex protocols written in TLA$^+$ to be tractable within reasonable resource constraints.

In Chapter 4 the symbolic model checking approach for Solidity of Marescotti et al. [2020] was evaluated and extended, with the goal of analysing and improving it. The evaluated approach is based on an encoding of Solidity into constrained Horn clauses which allows for unbounded model checking, but at the same time relies on a solving procedure that is undecidable in the general case. The results of the extensive experimentation done showed that the approach can establish safety and detect vulnerabilities in many real world smart contracts and outperforms comparable tools. In addition to the evaluation, an extension of the encoding to improve counterexample generation was also proposed.

In Chapter 5 a novel format for SMT unsatisfiability proofs was proposed, with the goal of producing compact proofs. The format builds on the DRAT format for unsatisfiability proofs for formulas in propositional logic by extending

it with proofs certificates for QF_LRA, QF_LIA, and QF_UF. The SMT solver OpenSMT was instrumented to produce proofs in the format and an independent proof checker capable of consuming proofs in the format, called TSWC, was implemented. The evaluation indicates that the format leads to smaller proofs that can be produced with a low overhead during solving, relative to other proof producing solvers, and can be efficiently checked.

In Chapter 6 a novel proof-backed approach for the validation of constrained Horn clauses models was proposed, with the goal of providing additional guarantees about the results obtained. The approach relies on proof producing SMT solvers, and the proof checker for their respective proof formats, to strengthen the guarantees provided by model validation. In addition, the approach is generic w.r.t. FOL theories and solvers, and is also very modular, enabling different SMT solvers to be used in the validation. The evaluation framework ATHENA was developed to asses the approach, in special the effect of different solver combinations. The results obtained indicate that using SMT proofs does not add significant overhead to model validation, with the sizes of the models themselves being a more significant obstacle to practical use.

The two main observations that can be made from the dissertation's results are, first, that applying formal verification to blockchain technologies can be made both practical and scalable, and, second, that the validation of logic solvers results can be efficiently done in an automated and lightweight manner.

Many interesting research avenues stem from the work presented in this dissertation. For the model checking approach for TLA$^+$, its enhancement to encode the remaining data structures in a structure preserving way, together with implementation support to cater for additional back-end solvers, are potential direct follow-ups. Another direction, which might require substantially more effort, is the development of an efficient unbounded model checking approach for TLA$^+$. For the model checking approach for Solidity, direct follow-ups include the enhancement of the encoding to handle scenarios such as calls to known external code and its instantiation to languages other than Solidity. A complementary direction is the embedding of CHC witnesses into the model checking approach. For the SMT unsatisfiability proofs, the addition of support for other theories is a clear direction. For the CHC model validation, the support for other theories is also a clear direction, with a complementary direction being the design of an approach for the validation of CHC unsatisfiability proofs. Lastly, the embedding of witness production and validation into verification tooling, both in the case of proofs and models, is a direction as interesting as it is unexplored.

Appendix A

Definition of the KerA$^+$ Language

The syntax of the KerA$^+$ language proposed by Konnov et al. [2019] and discussed in Section 3.1.1 is shown in Table A.1. The operators derived from TLA$^+$ preserve their semantics. The semantics of the non-TLA$^+$ based operators are described in the bullet points below.

Table A.1. Syntax of the KerA$^+$ language. The operators that do not have a counterpart in pure TLA$^+$ are highlighted.

Literals:	*true*, *false*	0,1,-1,2,-2,...	$c_1, \ldots, c_n$ (constants)	
Integers:	$i_1 \bullet i_2$ where $\bullet$ is one of	$+, -, *, \div, \%, <, \leq, >, \geq, =, \neq$		
Sets:	$\{e_1, \ldots, e_n\}$	$\{x \in S : p\}$	$\{e : x \in S\}$	UNION S
	$i_1 \,..\, i_2$	$Cardinality(S)$	$x \in [S_1 \to S_2]$	$x \in$ subset S
Control:	ITE(p, e_1, e_2)			
	$e_1 \oplus \ldots \oplus e_n$	$x' \in S$	$x' \in [S_1 \to S_2]$	$x' \in$ subset S
Quantifiers:	$\exists x \in S \,.\, p$	choose $x \in S : p$	FROM $e_1, \ldots, e_n$ BY θ	
Functions:	$[x \in S \mapsto e]$	$f[e]$	DOMAIN f	$[f$ except $![e_1] = e_2]$
Records:	$[nm_1 \mapsto e_1, \ldots, nm_n \mapsto e_n]$		DOMAIN r	$e.nm$
Tuples:	$\langle e_1, \ldots, e_n \rangle$	$t[i]$	DOMAIN t	
Sequences:	$\langle e_1, \ldots, e_n \rangle$	$s[i]$	DOMAIN s	$[s$ except $![i] = e]$
	$Len(s)$	$s \circ t$	$Head(s)$, $Tail(s)$	$SubSeq(s, i, j)$

- *Assignments* $x' \in S$, $x' \in [S_1 \to S_2]$, and $x' \in$ subset S. Following TLC, under the conditions given by Kukovec et al. [2018], an expression $x' \in S$ is treated as an assignment of a value from the set S to the variable x'. Note that an expression $x' = e$ is a special case of this rule, which can be written as $x' \in \{e\}$. Such assignments are labelled with $x' \in S$, to distinguish them from membership tests $x' \in S$.

- *Nondeterministic disjunction* $\phi_1 \oplus \ldots \oplus \phi_n$. This operator formalises TLC's special disjunction. It evaluates to *true* iff the disjunction $\phi_1 \vee \cdots \vee \phi_n$ evaluates to *true*. Nondeterministic disjunction, however, adds constraints on the variable assignments, i.e., for every $i, j \in 1..n$ and $i \neq j$, formula ϕ_i contains an assignment to a variable x' iff formula ϕ_j also contains an assignment to x'.

- *Choice with an oracle* $\mathsf{FROM}\ e_1, \ldots, e_n\ \mathsf{BY}\ \theta$. This operator returns expression e_i when $\theta = i$ and $1 \leq i \leq n$. Otherwise, it returns an arbitrary value of the same type as $e_1, \ldots, e_n$.

Appendix B

ARS Rules as Inferences

All reduction rules described in Chapter 3 are shown below as inferences. Complex rules require the stacking of premisses and may have other inferences as premisses. If multiple arenas are present in a rule $\rightarrow_{\mathcal{A}}$ is used to represent an edge in the arena $\mathcal{A}$.

- Inference rule for integer literal reduction, described in Section 3.1.2.

$$\frac{\left\langle num : \mathsf{Int} \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \qquad num \text{ is one of } 0, 1, -1, ...}{\left\langle \mathrm{c}_{num} \mid \mathcal{A}, \mathrm{c}_{num} : \mathsf{Int} \mid \nu \mid \Phi, \mathrm{c}_{num} = num \right\rangle} \text{ (INT)}$$

- Inference rule for picking, described in Section 3.1.2.

$$\frac{\left\langle \mathsf{FROM}\ \mathrm{c}_1, ..., \mathrm{c}_n\ \mathsf{BY}\ \theta : \tau \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \qquad \begin{array}{c} \mathrm{c}_1 : \tau, ..., \mathrm{c}_n : \tau \\ \tau \text{ is basic} \end{array}}{\left\langle \mathrm{c}_{pick} \mid \mathcal{A}, \mathrm{c}_{pick} : \tau \mid \nu \mid \Phi, \bigwedge_{1 \leq i \leq n} (\theta = i \rightarrow \mathrm{c}_{pick} = \mathrm{c}_i) \right\rangle} \text{ (FROMBASIC)}$$

- Inference rule for branching, described in Section 3.1.2.

$$\frac{\dfrac{\left\langle \mathsf{ITE}(\mathrm{c}_p, \mathrm{c}_1, \mathrm{c}_2) : \tau \mid \mathcal{A}_1 \mid \nu_1 \mid \Phi_1 \right\rangle \quad \left\langle \mathsf{FROM}\ \mathrm{c}_1, \mathrm{c}_2\ \mathsf{BY}\ \theta : \tau \mid \mathcal{A}_1, \theta : \mathsf{Int} \mid \nu_1 \mid \Phi_1, 1 \leq \theta \leq 2 \right\rangle}{\left\langle \mathrm{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2 \right\rangle}}{\left\langle \mathrm{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2, \theta = 1 \leftrightarrow \mathrm{c}_p \right\rangle} \text{ (ITE)}$$

- Inference rule for set enumeration, described in Section 3.2.1.

$$\frac{\left\langle \{\mathrm{c}_1, \ldots, \mathrm{c}_n\} : \mathsf{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle}{\left\langle \mathrm{c}_{set} \mid \mathcal{A}, \mathrm{c}_{set} : \mathsf{Set}[\tau], \mathrm{c}_{set} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n \mid \nu \mid \Phi, \mathit{EnumCtr} \right\rangle} \text{ (ENUM)}$$

- Inference rule for set filter, described in Section 3.2.1.

$$\frac{\begin{array}{c}\left\langle \{x \in \mathrm{c}_S : p\} : \mathsf{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \\ \mathrm{c}_S \rightarrow_{\mathcal{A}} \mathrm{c}_1, \ldots, \mathrm{c}_n \\ \dfrac{\left\langle p[\mathrm{c}_1/x] : \mathsf{Bool}, \ldots, p[\mathrm{c}_n/x] : \mathsf{Bool} \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle}{\left\langle \mathrm{c}_1^p, \ldots, \mathrm{c}_n^p \mid \mathcal{A}' \mid \nu' \mid \Phi' \right\rangle}\end{array}}{\left\langle \mathrm{c}_F \mid \mathcal{A}', \mathrm{c}_F : \mathsf{Set}[\tau], \mathrm{c}_F \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n \mid \nu' \mid \Phi', \mathit{FilterCtr} \right\rangle} \text{ (FILTER)}$$

- Inference rule for set map, described in Section 3.2.1.

$$\frac{\begin{array}{c}\left\langle \{e : x \in \mathrm{c}_S\} : \mathsf{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \\ \mathrm{c}_S \rightarrow_{\mathcal{A}} \mathrm{c}_1, \ldots, \mathrm{c}_n \\ \dfrac{\left\langle e[\mathrm{c}_1/x] : \tau, \ldots, e[\mathrm{c}_n/x] : \tau \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle}{\left\langle \mathrm{c}_1^e :, \ldots, \mathrm{c}_n^e \mid \mathcal{A}' \mid \nu' \mid \Phi' \right\rangle}\end{array}}{\left\langle \mathrm{c}_M \mid \mathcal{A}', \mathrm{c}_M : \mathsf{Set}[\tau], \mathrm{c}_M \rightarrow \mathrm{c}_1^e, \ldots, \mathrm{c}_n^e \mid \nu' \mid \Phi', \mathit{MapCtr} \right\rangle} \text{ (MAP)}$$

- Inference rule for function application, described in Section 3.2.2.

$$\frac{\begin{array}{c}\left\langle \mathrm{c}_{F_1}[\mathrm{c}_{arg}] : \tau \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \\ \mathrm{c}_F \xrightarrow[2]{} {}_{\mathcal{A}} \mathrm{c}_{F_{dom}} \rightarrow_{\mathcal{A}} \mathrm{c}_1^d, \ldots, \mathrm{c}_n^d \\ \mathrm{c}_F \rightarrow_{\mathcal{A}} \mathrm{c}_{F_{pairs}} \rightarrow_{\mathcal{A}} \mathrm{c}_1^p, \ldots, \mathrm{c}_n^p \\ \dfrac{\left\langle \mathsf{FROM}\ \mathrm{c}_1^p, \ldots, \mathrm{c}_n^p\ \mathsf{BY}\ \theta : \langle \tau_{arg}, \tau \rangle \mid \mathcal{A}, \theta : \mathsf{Int} \mid \nu \mid \Phi, 0 \leq \theta \leq n \right\rangle}{\left\langle \mathrm{c}^p \mid \mathcal{A}' \mid \nu' \mid \Phi' \right\rangle} \\ \mathrm{c}^p[2] \rightarrow_{\mathcal{A}'} \mathrm{c}_1, \ldots, \mathrm{c}_m\end{array}}{\left\langle \mathrm{c}_{res} \mid \mathcal{A}', \mathrm{c}_{res} : \tau, \mathrm{c}_{res} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_m \mid \nu' \mid \Phi', \mathit{FunAppCtr} \right\rangle} \text{ (FUNAPP)}$$

Appendix C

SMT Constraints Generated by Apalache's Constants Encoding

The constraints generated by APALACHE's constants encoding for the rules in Section 3.2 are shown below. Rule applications are also illustrated for both the arrays and the constants encodings.

Set Enumeration. The constraints added by the constants encoding to model set enumeration represent set membership and are Boolean constants of form $en\langle \mathrm{c}_{set}, i, \mathrm{c}_i \rangle$, with $1 \leq i \leq n$. Each constant explicitly represents an arena edge, with this approach encoding multisets due to the lack of a guarantee of uniqueness of the enumerated elements. The cell c_{set} itself is uninterpreted and remains unconstrained. The constants encoding version of *EnumCtr* is the following:

$$\bigwedge_{1 \leq i \leq n} en\langle \mathrm{c}_{set}, i, \mathrm{c}_i \rangle \qquad (EnumCtr_C)$$

The constraints generated by both encodings for the definition of set $Proc$ in Figure 2.1 are shown below, assuming we have a configuration containing three processes; $1..N$ is a shorthand for $\{1, ..., N\}$.

$$\begin{array}{l} a^0_{\mathrm{c}_{Proc}} = K_{\mathsf{Int}}(\bot) \wedge \\ a^1_{\mathrm{c}_{Proc}} = store(a^0_{\mathrm{c}_{Proc}}, \mathrm{c}_1, \top) \wedge \\ a^2_{\mathrm{c}_{Proc}} = store(a^1_{\mathrm{c}_{Proc}}, \mathrm{c}_2, \top) \wedge \\ a^3_{\mathrm{c}_{Proc}} = store(a^2_{\mathrm{c}_{Proc}}, \mathrm{c}_3, \top) \wedge \\ \mathrm{c}_{Proc} = a^3_{\mathrm{c}_{Proc}} \end{array} \qquad \begin{array}{l} en\langle \mathrm{c}_{Proc}, 1, \mathrm{c}_1 \rangle \wedge \\ en\langle \mathrm{c}_{Proc}, 2, \mathrm{c}_2 \rangle \wedge \\ en\langle \mathrm{c}_{Proc}, 3, \mathrm{c}_3 \rangle \end{array}$$

Set Membership. The constants encoding checks if c_x is equal to one of $c_1, ..., c_n$. The constants encoding version of the membership constraints shown in Section 3.2.1 are the following:

$$\bigvee_{1 \leq i \leq n} \underbrace{c_x = c_i}_{\text{cell equality}} \wedge \underbrace{en\langle c_{set}, i, c_i \rangle}_{\text{cell membership}} \qquad (MembershipCtr_C)$$

The constraints generated by both encodings for checking if process 2 in Figure 2.1 is correct are shown below.

$$c_{Corr}[c_2] \qquad \begin{array}{l} c_2 = c_1 \wedge en\langle c_{Corr}, 1, c_1 \rangle \vee \\ c_2 = c_2 \wedge en\langle c_{Corr}, 2, c_2 \rangle \vee \\ c_2 = c_3 \wedge en\langle c_{Corr}, 3, c_3 \rangle \end{array}$$

The arrays encoding adds a single constraint, while the constants encoding adds a linear amount of constraints and, due to the arena being an overapproximation, has to check not only that c_x is equal to one of the elements pointed to by c_{set}, but also that the Boolean constant encoding the edge to said element evaluates to *true*.

Set Filter. The constraints added by the constants encoding equate membership in F to membership in S and the predicate evaluation, as shown below. In a similar fashion to $EnumCtr_C$, and in contrast to *FilterCtr*, these constraints encode multisets and c_F remains unconstrained.

$$\bigwedge_{1 \leq i \leq n} en\langle c_F, i, c_i \rangle = (en\langle c_S, i, c_i \rangle \wedge c_i^p) \qquad (FilterCtr_C)$$

Invariant $NoDecide$ in Figure 2.1 can also be written using a set filter, as the expression $\{p \in Corr : pc[p] = \text{“}AC\text{”}\} = \emptyset$. The constraints generated by both encodings for this set filter are shown below.

$$\begin{array}{l} \text{ite}\big(\ c_1^p, a^0_{c_{ND}}[c_1], \neg a^0_{c_{ND}}[c_1]\ \big) \wedge \\ \text{ite}\big(\ c_2^p, a^0_{c_{ND}}[c_2], \neg a^0_{c_{ND}}[c_2]\ \big) \wedge \\ \text{ite}\big(\ c_3^p, a^0_{c_{ND}}[c_3], \neg a^0_{c_{ND}}[c_3]\ \big) \wedge \\ a^3_{c_{ND}} = map_\wedge(c_{Corr}, a^0_{c_{ND}}) \wedge \\ c_{ND} = a^3_{c_{ND}} \end{array} \qquad \begin{array}{l} en\langle c_{ND}, 1, c_1 \rangle = (en\langle c_{Corr}, 1, c_1 \rangle \wedge c_1^p) \wedge \\ en\langle c_{ND}, 2, c_2 \rangle = (en\langle c_{Corr}, 2, c_2 \rangle \wedge c_2^p) \wedge \\ en\langle c_{ND}, 3, c_3 \rangle = (en\langle c_{Corr}, 3, c_3 \rangle \wedge c_3^p) \end{array}$$

Both encodings generate a linear amount of constraints, since n $p[c_i/x]$ predicates have to be considered. Unlike is the case with *EnumCtr*, *FilterCtr* does not contain a sequence of store operations due to the usage map_f, which avoids the

creation of intermediary arrays; this is not possible in *EnumCtr* because it starts from a constant array.

Set Map. The constants encoding equates membership in M to membership in S. The constants encoding version of *MapCtr* is the following:

$$\bigwedge_{1 \le i \le n} en\langle \mathrm{c}_M, i, \mathrm{c}_i^e \rangle = en\langle \mathrm{c}_S, i, \mathrm{c}_i \rangle \qquad (MapCtr_C)$$

Invariant $NoDecide$ in Figure 2.1 can be written using a set map, as the expression "AC" $\notin \{pc[p] : p \in Corr\}$. The constraints generated by both encodings for this set map are shown below.

$$\begin{array}{l} a^0_{\mathrm{c}_{ND}} = K_{\mathsf{Int}}(\bot) \wedge \\ \mathrm{ite}\big(\ \mathrm{c}_{Corr}[\mathrm{c}_1], a^1_{\mathrm{c}_{ND}} = store(a^0_{\mathrm{c}_{ND}}, \mathrm{c}_1^e, \top), a^1_{\mathrm{c}_{ND}} = a^0_{\mathrm{c}_{ND}}\ \big) \wedge \\ \mathrm{ite}\big(\ \mathrm{c}_{Corr}[\mathrm{c}_2], a^2_{\mathrm{c}_{ND}} = store(a^1_{\mathrm{c}_{ND}}, \mathrm{c}_2^e, \top), a^2_{\mathrm{c}_{ND}} = a^1_{\mathrm{c}_{ND}}\ \big) \wedge \\ \mathrm{ite}\big(\ \mathrm{c}_{Corr}[\mathrm{c}_3], a^3_{\mathrm{c}_{ND}} = store(a^2_{\mathrm{c}_{ND}}, \mathrm{c}_3^e, \top), a^3_{\mathrm{c}_{ND}} = a^2_{\mathrm{c}_{ND}}\ \big) \wedge \\ \mathrm{c}_{ND} = a^3_{\mathrm{c}_{ND}} \end{array}$$

$$\begin{array}{l} en\langle \mathrm{c}_{ND}, 1, \mathrm{c}_1^e \rangle = en\langle \mathrm{c}_{Corr}, 1, \mathrm{c}_1 \rangle \wedge \\ en\langle \mathrm{c}_{ND}, 2, \mathrm{c}_2^e \rangle = en\langle \mathrm{c}_{Corr}, 2, \mathrm{c}_2 \rangle \wedge \\ en\langle \mathrm{c}_{ND}, 3, \mathrm{c}_3^e \rangle = en\langle \mathrm{c}_{Corr}, 3, \mathrm{c}_3 \rangle \wedge \end{array}$$

Function Definition. The constants encoding adds only one edge to the function cell c_F in the arena, $\mathrm{c}_F \xrightarrow{1} \mathrm{c}_{F_{pairs}}$. The cell $\mathrm{c}_{F_{pairs}}$ is fully encoded into SMT by the constants encoding, as the set of pairs $\{\langle x, f(x)\rangle : x \in \text{DOMAIN} f\}$. The constraints generated are thus the same as those of set map.

Function Domain. Accessing a function's domain in the constants encoding requires the domain set to be constructed by compiling the first element of each pair in $\mathrm{c}_{F_{pairs}}$. The constraints generated are thus the same as those of set map.

Function Update. The constants encoding treats the update as the expression $\{\mathrm{ite}(p[1] = arg, \langle arg, v\rangle, p) : p \in \mathrm{c}_{F_{pairs}}\}$, relying thus on set map; $p[1]$ stands for the first element of pair p.

Function Application. The constants encoding has the result cell simply as the second element of the pair chosen by the oracle, with any edges it might have already being present in $\mathcal{A}'$. Below we can see its version of rule FunApp, with

the constraints that are associated with it.

$$\begin{aligned} &\mathrm{c}_F[\mathrm{c}_{arg}] : \tau \text{ and } \mathrm{c}_F \xrightarrow{1} \mathrm{c}_{F_{pairs}} \rightarrow \mathrm{c}_1^p, \ldots, \mathrm{c}_n^p \\ &\quad \rightarrowtail \big(\text{ FROM } \mathrm{c}_1^p, \ldots, \mathrm{c}_n^p \text{ BY } \theta : \langle \tau_{arg}, \tau \rangle \mid \theta : \mathsf{Int} \mid 0 \leq \theta \leq n \rightarrowtail \mathrm{c}^p \big) \\ &\quad \rightarrowtail \mathrm{c}^p[2] \mid FunAppCtr_C \end{aligned} \tag{FunApp_C}$$

$$\bigwedge_{1 \leq i \leq n} \wedge \begin{array}{l} (\theta = i \rightarrow (\mathrm{c}_{arg} = \mathrm{c}_i^p[1] \wedge en\langle \mathrm{c}_{F_{pairs}}, i, \mathrm{c}_i^p \rangle) \\ (\theta = 0 \rightarrow (\mathrm{c}_{arg} \neq \mathrm{c}_i^p[1] \vee \neg en\langle \mathrm{c}_{F_{pairs}}, i, \mathrm{c}_i^p \rangle) \end{array} \tag{$FunAppCtr_C$}$$

For the function application $pc[p] =$ "*RD*" in line 34 of Figure 2.1 we have:

$$\begin{aligned} \mathrm{c}_{res} = \mathrm{c}_{pc}[\mathrm{c}_p] \qquad & (\theta = 1 \rightarrow (\mathrm{c}_p = \mathrm{c}_1^p[1] \wedge en\langle \mathrm{c}_{F_{pairs}}, 1, \mathrm{c}_1^p \rangle) \wedge \\ & (\theta = 2 \rightarrow (\mathrm{c}_p = \mathrm{c}_2^p[1] \wedge en\langle \mathrm{c}_{F_{pairs}}, 2, \mathrm{c}_2^p \rangle) \wedge \\ & (\theta = 3 \rightarrow (\mathrm{c}_p = \mathrm{c}_3^p[1] \wedge en\langle \mathrm{c}_{F_{pairs}}, 3, \mathrm{c}_3^p \rangle) \wedge \\ & (\theta = 0 \rightarrow (\mathrm{c}_p \neq \mathrm{c}_1^p[1] \vee \neg en\langle \mathrm{c}_{F_{pairs}}, 1, \mathrm{c}_1^p \rangle) \wedge \\ & (\theta = 0 \rightarrow (\mathrm{c}_p \neq \mathrm{c}_2^p[1] \vee \neg en\langle \mathrm{c}_{F_{pairs}}, 2, \mathrm{c}_2^p \rangle) \wedge \\ & (\theta = 0 \rightarrow (\mathrm{c}_p \neq \mathrm{c}_3^p[1] \vee \neg en\langle \mathrm{c}_{F_{pairs}}, 3, \mathrm{c}_3^p \rangle) \end{aligned}$$

Both encodings generate a linear amount of constraints in the general case, as shown by *FunAppCtr* and $FunAppCtr_C$. The arrays encoding, however, can have its constraints simplified to $\mathrm{c}_{res} = \mathrm{c}_F[\mathrm{c}_{arg}]$ when handling exclusively basic types, providing an efficiency benefit.

Bibliography

Albert, E., Correas, J., Gordillo, P., Román-Díez, G., and Rubio, A. (2019). SAFEVM: A Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, page 386–389.

Alt, L., Asadi, S., Chockler, H., Even Mendoza, K., Fedyukovich, G., Hyvärinen, A. E. J., and Sharygina, N. (2017). HiFrog: SMT-based Function Summarization for Software Verification. In *Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 207–213.

Alt, L., Blicha, M., Hyvärinen, A. E. J., and Sharygina, N. (2022). SolCMC: Solidity Compiler's Model Checker. In *Proceedings of the 34th International Conference on Computer Aided Verification*, pages 325–338.

Andoni, M., Robu, V., Flynn, D., Abram, S., Geach, D., Jenkins, D., McCallum, P., and Peacock, A. (2019). Blockchain Technology in the Energy Sector: A Systematic Review of Challenges and Opportunities. *Renewable and Sustainable Energy Reviews*, 100(1):143–174.

Andreotti, B., Lachnitt, H., and Barbosa, H. (2023). Carcara: An Efficient Proof Checker and Elaborator for SMT Proofs in the Alethe Format. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 367–386.

Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W., and Yellick, J. (2018). Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *Proceedings of the 13th EuroSys Conference*, pages 1–15.

Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., and Werner, B. (2011). A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Proceedings of the 1st International Conference on Certified Programs and Proofs*, pages 135–150.

Asadi, S., Blicha, M., Hyvärinen, A., Fedyukovich, G., and Sharygina, N. (2020). Incremental Verification by SMT-based Summary Repair. In *Proceedings of the 20th Conference on Formal Methods in Computer-Aided Design*, pages 77–82.

Atzei, N., Bartoletti, M., and Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts. In *Proceedings of the 6th International Conference on Principles of Security and Trust*, page 164–186.

Baek, S., Carneiro, M., and Heule, M. J. H. (2021). A Flexible Proof Format for SAT Solver-Elaborator Communication. In *Proceedings of the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 59–75.

Balint, A., Belov, A., Heule, M. J. H., and Järvisalo, M. (2013). SAT-COMP 2013: Competition Report. `http://hdl.handle.net/10138/40026`.

Bansal, K., Reynolds, A., Barrett, C., and Tinelli, C. (2016). A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In *Proceedings of the 8th International Joint Conference on Automated Reasoning*, pages 82–98.

Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. (2022a). CVC5: A Versatile and Industrial-Strength SMT Solver. In *Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442.

Barbosa, H., Blanchette, J. C., Fleury, M., and Fontaine, P. (2020). Scalable Fine-Grained Proofs for Formula Processing. *Journal of Automated Reasoning*, 64(3):485–510.

Barbosa, H., Blanchette, J. C., and Fontaine, P. (2017). Scalable Fine-Grained Proofs for Formula Processing. In *Proceedings of the 26th International Conference on Automated Deduction*, pages 398–412.

Barbosa, H., Hoenicke, J., and Bobot, F. (2022b). SMT-COMP 2022: Competition Report. `https://smt-comp.github.io/2022/slides-smtworkshop.pdf`.

Barnett, M., Chang, B.-Y. E., DeLine, R., Jacobs, B., and Leino, K. R. M. (2006). Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*, pages 364–387.

Barrett, C., Conway, C. L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., and Tinelli, C. (2011). CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 171–177.

Barrett, C., de Moura, L., and Fontaine, P. (2015). Proofs in Satisfiability Modulo Theories. *All about Proofs, Proofs for All*, 55(1):23–44.

Barrett, C., Fontaine, P., and Tinelli, C. (2021). The SMT-LIB Standard: Version 2.6. `https://smtlib.cs.uiowa.edu/language.shtml`.

Barrett, K., Cassels, B., Haahr, P., Moon, D. A., Playford, K., and Withington, P. T. (1996). A Monotonic Superclass Linearization for Dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, page 69–82.

Berkovits, I., Lazic, M., Losa, G., Padon, O., and Shoham, S. (2019). Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *Proceedings of the 31st International Conference on Computer Aided Verification*, pages 245–266.

Bertot, Y. and Castéran, P. (2004). *Interactive Theorem Proving and Program Development*. Springer Berlin, Heidelberg.

Besson, F., Fontaine, P., and Théry, L. (2011). A Flexible Proof Format for SMT: a Proposal. In *Proceedings of the 1st Workshop on Proof eXchange for Theorem Proving*, pages 15–26.

Beyer, D. (2023). Competition on Software Verification and Witness Validation: SV-COMP 2023. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 495–522.

Beyer, D. and Strejček, J. (2022). Case Study on Verification-Witness Validators: Where We Are and Where We Go. In *Proceedings of the 29th International Symposium on Static Analysis*, pages 160–174.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., and

Zanella-Béguelin, S. (2016). Formal Verification of Smart Contracts. In *Proceedings of the 11th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, page 91–96.

Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207.

Bjørner, N., Gurfinkel, A., McMillan, K., and Rybalchenko, A. (2015). Horn Clause Solvers for Program Verification. *Fields of Logic and Computation II. Lecture Notes in Computer Science*, 9300:24–51.

Blanchette, J. C., Böhme, S., Fleury, M., Smolka, S. J., and Steckermeier, A. (2016). Semi-Intelligible Isar Proofs from Machine-Generated Proofs. *Journal of Automated Reasoning*, 56(2):155–200.

Blass, A. and Gurevich, Y. (1987). Existential Fixed-Point Logic. *Lecture Notes in Computer Science*, 270:20–36.

Blicha, M., Britikov, K., and Sharygina, N. (2023). The Golem Horn Solver. In *Proceedings of the 35th International Conference on Computer Aided Verification*, pages 209–223.

Böhme, S. and Weber, T. (2010). Fast LCF-Style Proof Reconstruction for Z3. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*, pages 179–194.

Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., and Fontaine, P. (2009). veriT: An Open, Trustable and Efficient SMT-Solver. In *Proceedings of the 22nd International Conference on Automated Deduction*, pages 151–156.

Bracha, G. and Toueg, S. (1985). Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM*, 32(4):824–840.

Bradley, A. R. (2011). SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 70–87.

Brent, L., Grech, N., Lagouvardos, S., Scholz, B., and Smaragdakis, Y. (2020). Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 454–469.

Bruttomesso, R., Pek, E., Sharygina, N., and Tsitovich, A. (2010). The OpenSMT Solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153.

Bryant (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691.

Burch, J., Clarke, E., McMillan, K., Dill, D., and Hwang, L. (1992). Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170.

Bury, G. (2021). Dolmen: A Validator for SMT-LIB and Much More. In *Proceedings of the 19th International Workshop on Satisfiability Modulo Theories*, pages 32–39.

Buterin, V. (2014). Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform. `https://ethereum.org/en/whitepaper`.

Cachin, C. and Vukolic, M. (2017). Blockchain Consensus Protocols in the Wild. In *Proceedings of the 31st International Symposium on Distributed Computing*, pages 1–16.

Cadar, C., Dunbar, D., and Engler, D. (2008). KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, page 209–224.

Calzavara, S., Grishchenko, I., and Maffei, M. (2016). HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In *Proceedings of the 1st IEEE European Symposium on Security and Privacy*, pages 47–62.

Carter, M., He, S., Whitaker, J., Rakamarić, Z., and Emmi, M. (2016). SMACK Software Verification Toolchain. In *Proceedings of the 38th IEEE/ACM International Conference on Software Engineering*, pages 589–592.

Chaudhuri, K., Doligez, D., Lamport, L., and Merz, S. (2010). The TLA+ Proof System: Building a Heterogeneous Verification Platform. In *Proceedings of the 7th International Colloquium on the Theoretical Aspects of Computing*, pages 44–44.

Chen, H., Pendleton, M., Njilla, L., and Xu, S. (2020). A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses. *ACM Computing Surveys*, 53(3):1–43.

Christ, J., Hoenicke, J., and Nutz, A. (2012). SMTInterpol: An Interpolating SMT Solver. In *Proceedings of the 19th International SPIN Workshop*, pages 248–254.

Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R. (2018). *Handbook of Model Checking*. Springer International Publishing.

Clarke, E. M., Klieber, W., Nováček, M., and Zuliani, P. (2012). *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin, Heidelberg.

ConsenSys (2021). Mythril. `https://github.com/ConsenSys/mythril`.

Cruz-Filipe, L., Heule, M. J. H., Hunt, W. A., Kaufmann, M., and Schneider-Kamp, P. (2017). Efficient Certified RAT Verification. In *Proceedings of the 26th International Conference on Automated Deduction*, pages 220–236.

de Angelis, E. and Govind V. K., H. (2022). CHC-COMP 2022: Competition Report. In *Proceedings of the 9th Workshop on Horn Clauses for Verification and Synthesis*, pages 44–62.

de Moura, L. and Bjørner, N. (2008a). Proofs and Refutations, and Z3. In *Proceedings of the 7th International Workshop on the Implementation of Logics*, pages 123–132.

de Moura, L. and Bjørner, N. (2008b). Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340.

de Moura, L. and Bjørner, N. (2009). Generalized, Efficient Array Decision Procedures. In *Proceedings of the 9th Conference on Formal Methods in Computer-Aided Design*, pages 45–52.

Dobre, D. and Suri, N. (2006). One-step Consensus with Zero-Degradation. In *Proceedings of the 36th International Conference on Dependable Systems and Networks*, pages 137–146.

Egelund-Müller, B., Elsman, M., Henglein, F., and Ross, O. (2017). Automated Execution of Financial Contracts on Blockchains. *Business & Information Systems Engineering*, 59(6):457–467.

Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., and Barrett, C. (2017). SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In *Proceedings of the 29th International Conference on Computer Aided Verification*, pages 126–133.

Ernst, G. (2023). Korn - Software Verification with Horn Clauses (Competition Contribution). In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 559–564.

Fedyukovich, G., Sery, O., and Sharygina, N. (2013). eVolCheck: Incremental Upgrade Checker for C. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–307.

Feist, J., Grieco, G., and Groce, A. (2019). Slither: A Static Analysis Framework For Smart Contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, page 8–15.

Fontaine, P., Marion, J.-Y., Merz, S., Nieto, L. P., and Tiu, A. (2006). Expressiveness + Automation + Soundness: Towards Combining SMT Solvers and Interactive Proof Assistants. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 167–181.

Frank, J., Aschermann, C., and Holz, T. (2020). ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proceedings of the 29th USENIX Security Symposium*, page 2757–2774.

Gario, M. and Micheli, A. (2015). PySMT: a Solver-agnostic Library for Fast Prototyping of SMT-based Algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories*, pages 1–10.

Goel, A. and Sakallah, K. A. (2021a). On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In *Proceedings of the 13th NASA Formal Methods International Symposium*, page 131–150.

Goel, A. and Sakallah, K. A. (2021b). Towards an Automatic Proof of Lamport's Paxos. In *Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design*, pages 112–122.

Gordon, W. J. and Catalini, C. (2018). Blockchain Technology for Healthcare: Facilitating the Transition to Patient-Driven Interoperability. *Computational and Structural Biotechnology Journal*, 16(1):224–230.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. (2012). Synthesizing Software Verifiers from Proof Rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 405–416.

Guerraoui, R. (2001). On the Hardness of Failure-Sensitive Agreement Problems. *Information Processing Letters*, 79(2):99–104.

Gurfinkel, A. and Bjørner, N. (2019). The Science, Art, and Magic of Constrained Horn Clauses. In *Proceedings of the 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, page 6–10.

Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. (2015). The SeaHorn Verification Framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*, pages 343–361.

Hajdu, Á. and Jovanovic, D. (2020). Solc-Verify: A Modular Verifier for Solidity Smart Contracts. In *Proceedings of the 11th International Conference on Verified Software: Theories, Tools, and Experiments*, pages 161–179.

Heule, M., Hunt, W., Kaufmann, M., and Wetzler, N. (2017). Efficient, Verified Checking of Propositional Proofs. In *Proceedings of the 8th International Conference on Interactive Theorem Proving*, pages 269–284.

Heule, M. J. H., Hunt, W. A., and Wetzler, N. (2013a). Trimming While Checking Clausal Proofs. In *Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design*, pages 181–188.

Heule, M. J. H., Hunt, W. A., and Wetzler, N. (2013b). Verifying Refutations with Extended Resolution. In *Proceedings of the 24th International Conference on Automated Deduction*, pages 345–359.

Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., and Roşu, G. (2018). KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *Proceedings of the 31st IEEE Computer Security Foundations Symposium*, pages 204–217.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580.

Hoenicke, J. and Schindler, T. (2022). A Simple Proof Format for SMT. In *Proceedings of the 20th International Workshop on Satisfiability Modulo Theories*, pages 54–70.

Hojjat, H., Rümmer, P., Subotic, P., and Yi, W. (2014). Horn Clauses for Communicating Timed Systems. In *Proceedings of the 1st Workshop on Horn Clauses for Verification and Synthesis*, pages 39–52.

Hojjat, H. and Rümmer, P. (2018). The Eldarica Horn Solver. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design*, pages 1–7.

Jiao, J., Kan, S., Lin, S.-W., Sanan, D., Liu, Y., and Sun, J. (2020). Semantic Understanding of Smart Contracts: Executable Operational Semantics of Solidity. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 1695–1712.

K Framework (2018a). KSolidity: Semantics of Solidity in K. `https://github.com/kframework/solidity-semantics`.

K Framework (2018b). KVyper: Semantics of Vyper in K. `https://github.com/kframework/vyper-semantics`.

Kahsai, T., Rümmer, P., Sanchez, H., and Schäf, M. (2016). JayHorn: A Framework for Verifying Java programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*, pages 352–358.

Kalra, S., Goel, S., Dhawan, M., and Sharma, S. (2018). ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, pages 1–15.

Katz, G., Barrett, C., Tinelli, C., Reynolds, A., and Hadarean, L. (2016). Lazy Proofs for DPLL(T)-based SMT Solvers. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 93–100.

Komuravelli, A., Gurfinkel, A., and Chaki, S. (2016). SMT-based Model Checking for Recursive Programs. *Formal Methods in System Design*, 48(3):175–205.

Konnov, I., Kukovec, J., and Tran, T.-H. (2019). TLA+ Model Checking Made Symbolic. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

Konnov, I., Kuppe, M., and Merz, S. (2022). Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In *Proceedings of the 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 88–105.

Konnov, I., Lazić, M., Stoilkovska, I., and Widder, J. (2020). Tutorial: Parameterized Verification with Byzantine Model Checker. In *Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 189–207.

Kroening, D. and Strichman, O. (2016). *Decision Procedures - An Algorithmic Point of View*. Springer Berlin, Heidelberg.

Kukovec, J., Tran, T.-H., and Konnov, I. (2018). Extracting Symbolic Transitions from TLA+ Specifications. In *Proceedings of the 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 89–104.

Kumar, R., Myreen, M. O., Norrish, M., and Owens, S. (2014). CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 179–191.

Lammich, P. (2020). Efficient Verified (UN)SAT Certificate Checking. *Journal of Automated Reasoning*, 64(3):513–532.

Lamport, L. (2002). *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc.

Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, pages 75–86.

Lincoln, P. and Rushby, J. (1993). A Formally Verified Algorithm for Interactive Consistency under a Hybrid Fault Model. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, pages 402–411.

LLVM Team (2017). libFuzzer – A Library for Coverage-Guided Fuzz Testing. `https://llvm.org/docs/LibFuzzer.html`.

Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. (2016). Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, page 254–269.

Marescotti, M., Blicha, M., Hyvärinen, A. E. J., Asadi, S., and Sharygina, N. (2018). Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *Proceedings of the 8th International Symposium on Leveraging Applications of Formal Methods*, pages 450–465.

Marescotti, M., Otoni, R., Alt, L., Eugster, P., Hyvärinen, A. E. J., and Sharygina, N. (2020). Accurate Smart Contract Verification Through Direct Modelling. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 178–194.

Marmsoler, D. and Brucker, A. D. (2021). A Denotational Semantics of Solidity in Isabelle/HOL. In *Proceedings of the 19th International Conference on Software Engineering and Formal Methods*, pages 403–422.

Marques-Silva, J. P. and Sakallah, K. A. (1999). GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506–521.

Matsushita, Y., Tsukada, T., and Kobayashi, N. (2021). RustHorn: CHC-Based Verification for Rust Programs. *ACM Transactions on Programming Languages and Systems*, 43(4):1–54.

McMillan, K. L. and Padon, O. (2020). Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *Proceedings of the 32nd International Conference on Computer Aided Verification*, page 190–202.

Merz, S. and Vanzetto, H. (2018). Encoding TLA+ into Unsorted and Many-Sorted First-Order Logic. *Science of Computer Programming*, 158:3–20.

Min, T., Wang, H., Guo, Y., and Cai, W. (2019). Blockchain Games: A Survey. In *Proceedings of the 2019 IEEE Conference on Games*, pages 1–8.

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., and Dinaburg, A. (2019). Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, pages 1186–1189.

Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. `http://bitcoin.org/bitcoin.pdf`.

Nehaï, Z. and Bobot, F. (2020). Deductive Proof of Industrial Smart Contracts Using Why3. In *Proceedings of the 2019 Formal Methods International Workshops*, pages 299–311.

Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. (2015). How Amazon Web Services uses Formal Methods. *Communications of the ACM*, 58(4):66–73.

Nieuwenhuis, R. and Oliveras, A. (2005). Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Rewriting Techniques and Applications*, pages 453–468.

Nikolić, I., Kolluri, A., Sergey, I., Saxena, P., and Hobor, A. (2018). Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, page 653–663.

Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A. E. J., and Sharygina, N. (2021). Theory-Specific Proof Steps Witnessing Correctness of SMT Executions. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*, pages 541–546.

Otoni, R., Blicha, M., Eugster, P., and Sharygina, N. (2023a). CHC Model Validation with Proof Guarantees. Accepted to the *18th International Conference on integrated Formal Methods*.

Otoni, R., Konnov, I., Kukovec, J., Eugster, P., and Sharygina, N. (2023b). Symbolic Model Checking for TLA+ Made Faster. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 126–144.

Otoni, R., Marescotti, M., Alt, L., Eugster, P., Hyvärinen, A., and Sharygina, N. (2023c). A Solicitous Approach to Smart Contract Verification. *ACM Transactions on Privacy and Security*, 26(2):1–28.

Ozdemir, A., Niemetz, A., Preiner, M., Zohar, Y., and Barrett, C. (2019). DRAT-based Bit-Vector Proofs in CVC4. In *Proceedings of the 22nd International Conference on Theory and Applications of Satisfiability Testing*, pages 298–305.

Padon, O., McMillan, K. L., Panda, A., Sagiv, M., and Shoham, S. (2016). Ivy: Safety Verification by Interactive Generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630.

Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., and Vechev, M. (2020). VerX: Safety Verification of Smart Contracts. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 1661–1677.

Reeves, J. E., Kiesl-Reiter, B., and Heule, M. J. H. (2023). Propositional Proof Skeletons. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 329–347.

Reynolds, J. C. (1974). Towards a Theory of Type Structure. In *Proceedings of the Programming Symposium*, page 408–423.

Rius, A. D. D. M. and Gashier, E. (2020). Smart Derivatives: On-Chain Forwards for Digital Assets. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods*, pages 195–211.

Rival, X. and Yi, K. (2020). *Introduction to Static Analysis*. MIT Press.

Roşu, G. and Şerbănuţă, T. F. (2010). An Overview of the K Semantic Framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434.

Rouhani, S. and Deters, R. (2019). Security, Performance, and Applications of Smart Contracts: A Systematic Survey. *IEEE Access*, 7(1):50759–50779.

Sandberg Ericsson, A., Myreen, M. O., and Åman Pohjola, J. (2019). A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning*, 63(2):463–488.

Schmidt, J. and Leuschel, M. (2021). Improving SMT Solver Integrations for the Validation of B and Event-B Models. In *Proceedings of the 26th International Conference on Formal Methods for Industrial Critical Systems*, pages 107–125.

Schneidewind, C., Grishchenko, I., Scherer, M., and Maffei, M. (2020). EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, page 621–640.

Schultz, W., Dardik, I., and Tripakis, S. (2022). Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*, pages 273–283.

Schurr, H.-J., Fleury, M., Barbosa, H., and Fontaine, P. (2021). Alethe: Towards a Generic SMT Proof Format. In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving*, pages 49–54.

Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., and Hao, K. C. G. (2019). Safer Smart Contract Programming with Scilla. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–30.

Sery, O., Fedyukovich, G., and Sharygina, N. (2012). FunFrog: Bounded Model Checking with Interpolation-Based Function Summarization. In *Proceedings of the 10th International Symposium on Automated Technology for Verification and Analysis*, pages 203–207.

Sinz, C. and Biere, A. (2006). Extended Resolution Proofs for Conjoining BDDs. In *Proceedings of the 1st International Symposium on Computer Science in Russia*, pages 600–611.

So, S., Lee, M., Park, J., Lee, H., and Oh, H. (2020). VeriSmart: A Highly Precise Safety Verifier for Ethereum Smart Contracts. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, pages 1678–1694.

Stephens, J., Ferles, K., Mariano, B., Lahiri, S., and Dillig, I. (2021). SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *Proceedings of the 42nd IEEE Symposium on Security and Privacy*, pages 555–571.

Stoilkovska, I., Konnov, I., Widder, J., and Zuleger, F. (2022). Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. *International Journal on Software Tools for Technology Transfer*, 24(1):33–48.

Stump, A., Barrett, C., Dill, D., and Levitt, J. (2001). A Decision Procedure for an Extensional Theory of Arrays. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, pages 29–37.

Stump, A., Oe, D., Reynolds, A., Hadarean, L., and Tinelli, C. (2013). SMT Proof Checking Using a Logical Framework. *Formal Methods in System Design*, 42(1):91–118.

Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.-Y., Kohlweiss, M., Zinzindohoue, J.-K., and Zanella-Béguelin, S. (2016). Dependent Types and Multi-monadic Effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 256–270.

Tan, B., Mariano, B., Lahiri, S. K., Dillig, I., and Feng, Y. (2022). SolType: Refinement Types for Arithmetic Overflow in Solidity. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29.

Tange, O. (2011). GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine*, 36(1):42–47.

Tran, T.-H. (2023). *Symbolic Verification of TLA+ Specifications with Applications to Distributed Algorithms*. PhD thesis, Technische Universität Wien. Upcoming thesis.

Tsankov, P., Dan, A., Drachsler-Cohen, D., Gervais, A., Bünzli, F., and Vechev, M. (2018). Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, page 67–82.

Van Gelder, A. (2012). Producing and Verifying Extremely Large Propositional Refutations - Have Your Cake and Eat it Too. *Annals of Mathematics and Artificial Intelligence*, 65(4):329–372.

van Steen, M. and Tanenbaum, A. S. (2017). *Distributed Systems*. CreateSpace Independent Publishing Platform.

Wang, X., Yang, W., Noor, S., Chen, C., Guo, M., and van Dam, K. H. (2019). Blockchain-Based Smart Contract for Energy Demand Management. *Energy Procedia*, 158(1):2719–2724.

Wang, Y., Lahiri, S. K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., and Ferles, K. (2020). Formal Verification of Workflow Policies for Smart Contracts in Azure Blockchain. In *Proceedings of the 11th International Conference on Verified Software: Theories, Tools, and Experiments*, pages 87–106.

Wesley, S., Christakis, M., Navas, J. A., Trefler, R., Wüstholz, V., and Gurfinkel, A. (2022). Verifying Solidity Smart Contracts via Communication Abstraction in SmartACE. In *Proceedings of the 23rd International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 425–449.

Wetzler, N., Heule, M. J. H., and Hunt, W. A. (2014). DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing*, pages 422–429.

Wood, G. (2015). Ethereum: A Secure Decentralised Generalised Transaction Ledger. `https://github.com/ethereum/yellowpaper`.

Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41(4):1–36.

Yu, Y., Manolios, P., and Lamport, L. (1999). Model Checking TLA+ Specifications. In *Proceedings of the 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66.

Zhang, P., White, J., Schmidt, D. C., Lenz, G., and Rosenbloom, S. T. (2018). FHIRChain: Applying Blockchain to Securely and Scalably Share Clinical Data. *Computational and Structural Biotechnology Journal*, 16(1):267–278.

Zilliqa Team (2017). Zilliqa Technical Whitepaper. `https://docs.zilliqa.com/whitepaper.pdf`.