CHC Model Validation with Proof Guarantees

Rodrigo Otoni[1✉], Martin Blicha[1,2], Patrick Eugster[1], and Natasha Sharygina[1]

[1] Università della Svizzera italiana, Lugano, Switzerland
{otonir,blichm,eugstp,sharygin}@usi.ch
[2] Charles University, Prague, Czech Republic

In the proceedings of iFM'23: https://doi.org/10.1007/978-3-031-47705-8_4

Abstract. Formal verification tooling increasingly relies on logic solvers as automated reasoning engines. A point of commonality among these solvers is the high complexity of their codebases, which makes bug occurrence disturbingly frequent. Tool competitions have showcased many examples of state-of-the-art solvers disagreeing on the satisfiability of logic formulas, be them solvers for Boolean satisfiability (SAT), satisfiability modulo theories (SMT), or constrained Horn clauses (CHC). The validation of solvers' results is thus of paramount importance, in order to increase the confidence not only in the solvers themselves, but also in the tooling which they underpin. Among the formalisms commonly used by modern verification tools, CHC is one that has seen, at the same time, extensive practical usage and very little efforts in result validation. As one of the initial steps in addressing this issue, we propose and evaluate a two-layered validation approach for witnesses of CHC satisfiability. Our approach relies, first, on a proof producing SMT solver to validate a CHC model via a series of SMT queries, and, second, on a proof checker to validate the SMT solver's results. We developed a modular evaluation framework and assessed the approach's viability via large scale experimentation, comparing three CHC solvers, five SMT solvers, and four proof checkers. Our results indicate that the approach is feasible, with potential to be incorporated into CHC-based tooling, and also confirm the need for validation, with nine bugs being found in the tools used.

Keywords: validation · constrained Horn clauses · SMT proofs

1 Introduction

First-order logic (FOL) is a formalism capable of representing many interesting verification problems, ranging from simple integer overflow to elaborate safety and liveness properties. Different fragments of FOL are suitable to aid in specific verification tasks, with one fragment of particular practical interest being constrained Horn clauses (CHC) [28]. The CHC fragment has been shown to be a match for Hoare logic [33] with practical uses [12], aiding in reasoning about the behaviour of procedural [12] and functional [27] programs, as well as concurrent systems [35] and smart contracts [46], to name a few examples.

To enable the automated reasoning of FOL formulas different logic solvers can be used, depending on the fragment selected. The most common categories

of such tools are arguably Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers [40], which respectively solve formulas in the propositional fragment of FOL and in extensions of it with theories such as arithmetics, arrays, and bit vectors. CHC solvers are also available, e.g., Eldarica [36], Golem [14], and Spacer [39], serving, for instance, as the back-end reasoning engines of verification tools targeting programs written in C/C++ [29], Java [38], Rust [43], and Solidity [1], as well as Android applications [18].

Despite their extensive usage in verification, logic solvers are themselves not immune to bugs. To illustrate this point, in the 2022 edition of the annual SMT competition there were 18 benchmarks in which at least two state-of-the-art solvers disagreed on the results [7][3]. 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) [41] and garbage collector [49] implementations. Despite 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 [4,20,30,32,51,54] and SMT [34,44,45,50,52] solvers, and both the annual SAT and SMT competitions now following this approach[4]. Codebase verification can still be applied, however, targeting instead the validation tools [31,42], which are much less complex and easier to maintain.

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 [21]. The input of a CHC solver, detailed in Section 2, 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, UNSAT for short, and if it cannot the input is considered satisfiable, SAT for short, not to be confused with Boolean satisfiability. 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; for the remainder of the paper UNSAT proofs and SAT models will be referred to simply as proofs and models.

[3] This is down from 274 benchmarks in the 2021 edition, showcasing the attention given by competitors to addressing unsound results found.

[4] The SAT competition requires its competitors to produce witnesses since its 2013 edition, while the SMT competition started an exhibition track for this, still separate from the main tracks, in its 2022 edition.

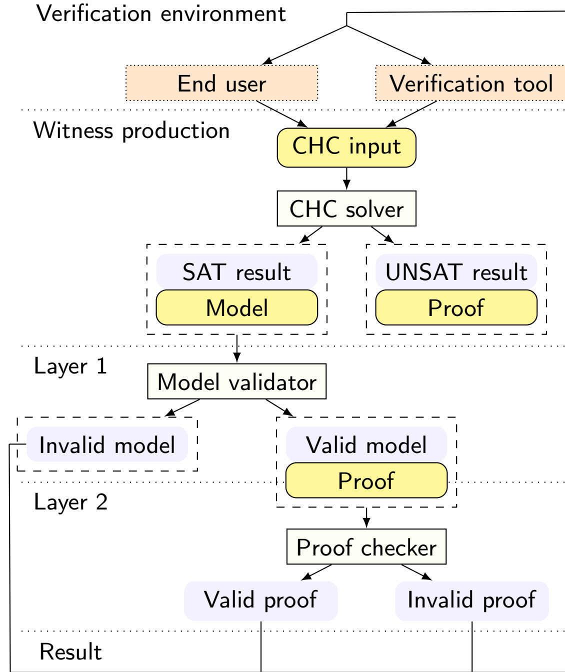


Fig. 1: Overview of our two-layered validation approach for CHC models. Although capable of being produced, CHC proofs cannot be checked currently.

The production of witnesses is a common feature of modern CHC solvers, but 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 [22][5]. Unlike the case for models, however, the proofs produced cannot be validated with available tooling, given that, to the best of our knowledge, 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, we propose a two-layered validation approach to provide additional guarantees about the results obtained, illustrated in Figure 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.

To asses the viability of practical model validation we developed a modular evaluation framework, called ATHENA, capable of catering to different combinations of state-of-the-art CHC and SMT solvers, and conducted a large scale evaluation. Concretely, we used our framework to validate the models produced by three CHC solvers, Eldarica [36], Golem [14], and Spacer [39], with each model produced being separately validated, in Layer 1, by five proof producing SMT solvers, cvc5 [5], OpenSMT [37], SMTInterpol [19], veriT [16], and Z3 [22]. In addition, we checked, in Layer 2, all the proofs produced in the proof formats currently supported by automated proof checkers, by using the checkers Carcara [2], LFSC checker [52], SMTInterpol checker [34], and TSWC [45].

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

Table 1: Brief descriptions of the bugs found during the evaluation.

		Bug Effect
CHC solvers	Eldarica	Invalid model produced[10]
	Spacer	Invalid model produced[12]
	Golem	Syntactically malformed model produced[9]
	Golem	Crash during model production[8]
SMT solvers	cvc5	Invalid proof produced[15]
	cvc5	Crash during proof production[14]
	OpenSMT	Crash during sort inference[13]
Proof checkers	Carcara	Parsing error due to unknown attribute[16]
	LFSC checker	Crash during type inference[17]

To have a focused evaluation we conducted our experiments on benchmarks from one specific FOL theory, namely the linear integer arithmetic (LIA) theory. We used all 955 LIA benchmarks from CHC-COMP 2022 in our 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 used led to 91626 SMT instances and 385303 SMT proofs being produced as part of 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 [29] and SolCMC [1] 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 our 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 [47], being a potential complementary goal. Lastly, model validation provides a useful way to generate new and interesting SMT instances. Our evaluation uncovered bugs not only in the selected CHC solvers, which are our main focus, but also in two SMT solvers and two proof checkers for SMT proofs. The bugs found, listed in Table 1, range from parsing errors to invalid models being produced. They were all acknowledged by the developers and are detailed in Section 5. In addition to aiding in tool development, these bugs confirm the need for additional guarantees to be provided to modern verification tooling.

To summarise, our contributions are the following:

1. Proposal of a two-layered validation approach for CHC models;
2. Development of an evaluation framework, ATHENA, to assess the approach;
3. Staging of a large scale evaluation to determine the viability of the approach.

The remainder of the paper is structured as follows: the necessary background is given in Section 2, the two layers of our validation approach are presented in Section 3, ATHENA is detailed in Section 4, the evaluation is discussed in Section 5, and, finally, our conclusions are laid out in Section 6.

2 Background

The constrained Horn clauses formalism has been proposed as a unified, purely logic-based, intermediate language for reasoning about verification tasks [27]. It builds upon the success achieved with SAT and SMT, using logical constraints to capture various verification tasks, including safety, termination, and loop invariant computation, from a variety of domains. In this section the necessary CHC background is presented, followed by an overview of related witness validation approaches. We refer readers to [40] for details on SAT and SMT.

2.1 Constrained Horn Clauses

Following standard SMT terminology [9], we assume a first-order theory $\mathcal{T}$ and a set of uninterpreted predicates $\mathcal{P}$ disjoint from the signature of $\mathcal{T}$. A constrained Horn clause is a formula $\varphi \land P_1 \land \ldots \land P_n \implies H$, where φ is an interpreted formula in the language of $\mathcal{T}$, each P_i is an application of a symbol $p \in \mathcal{P}$ to terms of $\mathcal{T}$, H is either an application of a symbol $p \in \mathcal{P}$ to terms of $\mathcal{T}$ or *false*, and all variables in the formula are implicitly universally quantified. Commonly, the antecedent and the consequent of the implication are denoted as the *body* and the *head* of the Horn clause, and φ is referred to as its *constraint*.

Given a set of constrained Horn clauses $\mathcal{S}$ over the uninterpreted predicates $\mathcal{P}$ and theory $\mathcal{T}$, we say that $\mathcal{S}$ is satisfiable if there exists a model $\mathcal{M}$ of $\mathcal{T}$ extended with an interpretation for all the uninterpreted predicates $\mathcal{P}$ such that all the clauses evaluate to *true* in $\mathcal{M}$, i.e., every clause evaluate to *true* in $\mathcal{M}$ for all possible instantiations of the universally quantified variables. In practice, we are interested in interpretations that are definable in the language of $\mathcal{T}$, i.e., we want a mapping of the predicates to a set of formulas in the language of $\mathcal{T}$ such that each clause from $\mathcal{S}$ is valid in $\mathcal{T}$ after the uninterpreted predicates are replaced by their interpretations. This is called *syntactic solvability*, as opposed to the more general *semantic solvability* [48]. The interpretations of the predicates from the discovered model serve as witnesses for the satisfiability answer. An unsatisfiability answer, on the other hand, needs to be witnessed by a derivation of *false* from the original clauses, likely via universal instantiation and resolution.

2.2 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 [51] and clausal proofs [30], with resolution asymmetric tautology (RAT) [32] being a base for many recent developments, e.g., deletion RAT (DRAT) [54], linear RAT (LRAT) [20], and flexible RAT (FRAT) [4]. The production of proofs in the DRAT format has been a requirement in the SAT competition since its 2014 edition, with DRAT-TRIM [54] 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 organizers having been made in recent years for the adoption of a unified format in line with the SMT-LIB standard [8]. A separate, experimental, model validation track has been established and has seen a steady increase in the SMT-LIB logics supported, with PySMT [26] and Dolmen [17] 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 [50] 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) [52], with LFSC support preceding Alethe's integration and dating back to the CVC3 version of the tool. SMTInterpol [34], OpenSMT [45], and Z3 [44] 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 [3,23] and Isabelle/HOL [6,13,15,25]. 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 [2,34,45,52,54]. 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 [10]. Software verification witnesses are different from those used by logic solvers, being categorizes as either correctness or violation witnesses, with their own formats[6] and limitations [11]. The tool that maybe best

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

illustrates usage of witness is Korn [24], 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.

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.

We propose a two-layered validation approach for CHC models, detailed in Figure 2; since our 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 model definition laid out in Section 2.1, model validation can be done via a number of SMT queries which is linear w.r.t. to the number of Horn clauses present in the input. Each such query checks if a specific Horn clause is logically valid in the theory $\mathcal{T}$ 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 $\varphi \wedge P_1 \wedge \ldots \wedge P_n \wedge \neg H$ exists. This check is well suited for

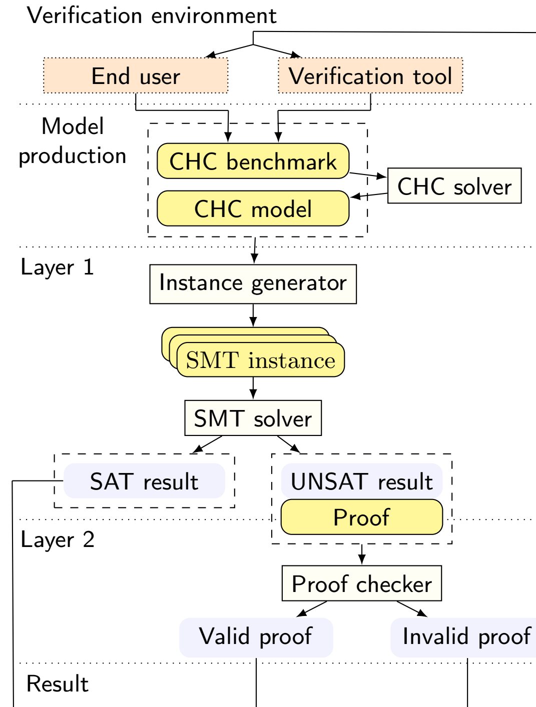


Fig. 2: Breakdown of our two-layered validation approach for CHC models. A valid model will have all the SMT instances generated from it yield an UNSAT result backed by a valid SMT proof.

SMT solving, with a valid model leading to all queries being unsatisfiable. An important note is that, depending on the theory $\mathcal{T}$, the query checking might be intractable for existing 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*:

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

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

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

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

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

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 approach we propose 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 we provide additional correctness guarantees for CHC model validation.

Our 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, our approach can also be embedded into CHC-based verification tools, enhancing their own guarantees.

4 Implementation

To enable the practical use of our 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. Our 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 [53] to achieve a large degree of parallelisation in order to better tackle the high computation cost. ATHENA is open-source[7], enabling third-parties to make full use of it, with one of our goals being to provide the groundwork for model validation at CHC-COMP.

5 Evaluation

We first describe the benchmarks and tools used, in Section 5.1, and then discuss the results obtained related to CHC model validation, in Section 5.2, and SMT proof checking, in Section 5.3. We used a machine with 64 AMD EPYC 7452 processors and 256 GB of memory for the evaluation. All individual tool executions had a timeout of 60 seconds and a memory limit of 5 gigabytes.

5.1 Benchmarks and Tools

We used the benchmarks of the two LIA tracks of CHC-COMP 2022 [21], the LIA-lin track, consisting of benchmarks containing only linear Horn clauses, and the LIA-nonlin track, consisting of benchmarks containing nonlinear Horn clauses. We 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 we chose the current three best performing CHC solvers in the LIA tracks for comparison, which are, in alphabetical order, ELDARICA [36], GOLEM [14], and SPACER [39]. For model validation we used all SMT solvers that competed in the proof exhibition track of the 2022 SMT competition, which are, in alphabetical order, CVC5 [5], OPENSMT [37], SMTINTERPOL [19], and VERIT [16], as well as Z3 [22], which can produce proofs but did not compete in the track. To check the SMT proofs we used the fully automated checkers CARCARA [2], for ALETHE proofs produced by CVC5 and VERIT, LFSC checker [52], for LFSC proofs produced by CVC5, SMTINTERPOL checker [34], for proofs produced by SMTINTERPOL, and TSWC [45], for proofs produced by OPENSMT; to the best of our knowledge there is currently no independent automated checker for proofs produced by Z3.

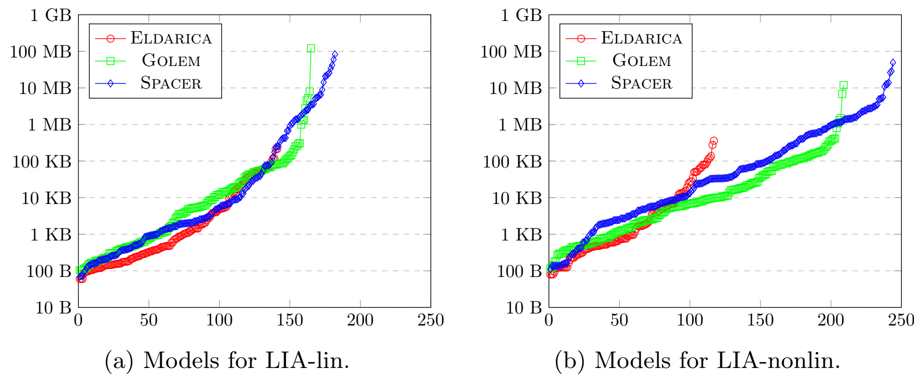
5.2 Model Validation Results

To produce the CHC models we executed the selected CHC solvers with all benchmarks; the results are summarised in Table 2. All tools were executed with

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

Table 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



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

Fig. 3: Sizes of the CHC models. The models are ordered according to their size, the x-axis indicates their position in the order and the y-axis indicates their size.

their default engine configurations. The performance of the tools is in line with the CHC-COMP results, with Spacer solving the most benchmarks overall, 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 our framework[8], as well as syntactically malformed models being produced by it[9], 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 be seen in Figure 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.

[8] See https://github.com/usi-verification-and-security/golem/issues/29.
[9] See https://github.com/usi-verification-and-security/golem/issues/27.

Table 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 SMT solver is not equipped to handle some features of the instance.

		SAT	UNSAT	UNK	TO	MO	ERR	UNS
LIA-lin Eldarica (5050 instances)	cvc5	0	5041	0	0	0	9	0
	OpenSMT	0	4970	0	0	0	80	0
	SMTInterpol	0	5041	0	0	0	9	0
	veriT	0	4986	0	0	0	9	55
	Z3	3	5047	0	0	0	0	0
LIA-lin Golem (5268 instances)	cvc5	0	5268	0	0	0	0	0
	OpenSMT	0	5268	0	0	0	0	0
	SMTInterpol	0	5265	0	3	0	0	0
	veriT	0	5216	0	0	0	0	52
	Z3	0	5268	0	0	0	0	0
LIA-lin Spacer (16232 instances)	cvc5	695	15464	0	73	0	0	0
	OpenSMT	7	700	0	0	0	912	14613
	SMTInterpol	105	11909	28	4190	0	0	0
	veriT	19	1543	0	0	0	0	14670
	Z3	690	15026	0	516	0	0	0

To validate each model we executed the SMT instances generated from it 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. We consider the SMT instances generated for the models produced by each CHC solver, by track, as separate instance sets, thus we have three instance sets per track. The results for the LIA-lin and LIA-nonlin instance sets can be seen in tables 3 and 4, respectively; 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[10], which leads to an error in most SMT solvers, but can be solved by Z3 via unit propagation[11]. Spacer's invalid models are due to problematic internal transformations[12]. Another aspect of model quality

[10] See https://github.com/uuverifiers/eldarica/issues/51.
[11] See https://github.com/Z3Prover/z3/issues/6719.
[12] See https://github.com/Z3Prover/z3/issues/6716.

Table 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 SMT 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

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[13], 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.

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 we executed the selected SMT solvers in proof production mode. The number of proofs produced by each SMT solver can be seen in Table 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 3 and 4. Concretely, the combined percentage of proofs produced in relation to the

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

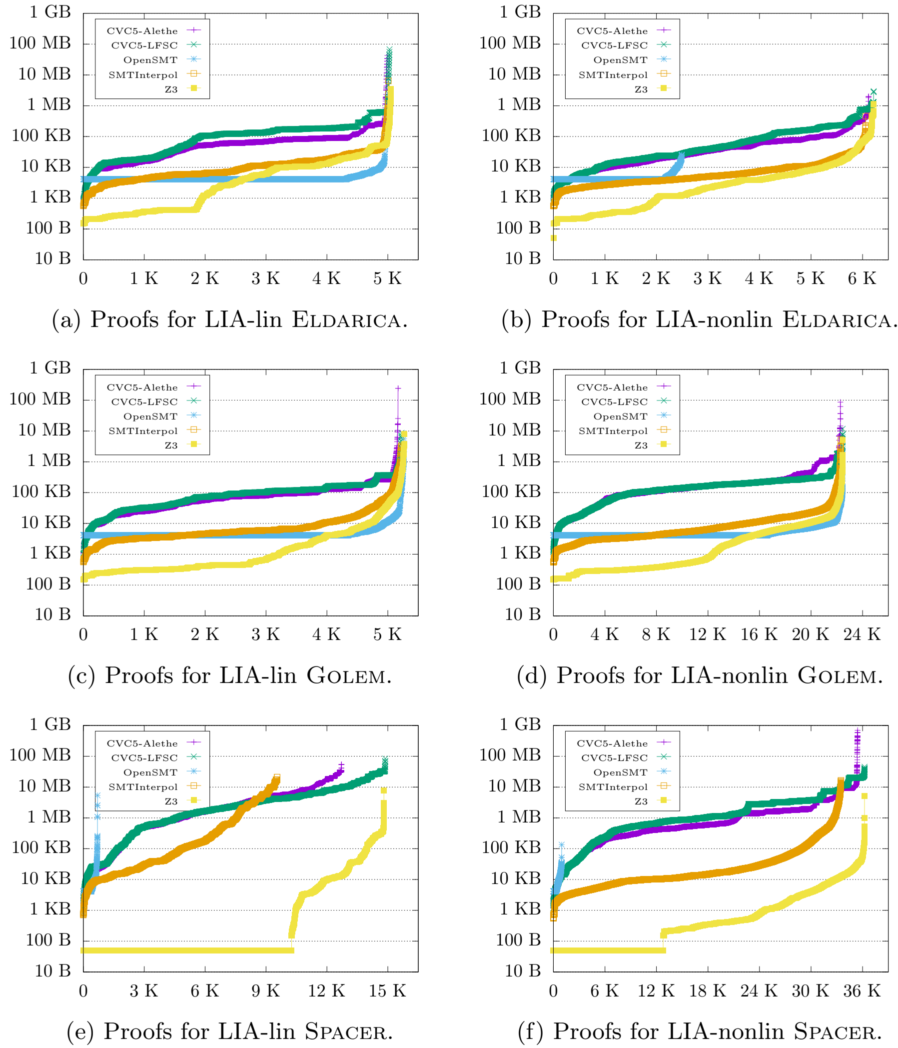
Table 5: Number of proofs produced by the selected SMT solvers; cvc5 has separate results for its two proof formats. Each column shows the amount of proofs produced from a given instance set, with the total amount of instances in each set shown below the CHC solver that produced the models for it.

	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

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 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[14].

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 our evaluation can be seen in Figure 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 simply 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

[14] 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.

Fig. 4: Sizes of the SMT proofs. 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.

and Spacer's 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 a size of 699 MB, which is a good illustration of the need of compact proofs.

Table 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 presented 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

To check the proofs we used the available automated checkers suitable for each proof format, 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 and 7, respectively. 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[15]. 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 number of errors when checking proofs produced for SMT instances generated from Spacer models, due to the presence of attribute annotations in models containing quantifiers[16]. Lastly, 3 errors were also observed with the LFSC checker, due to a type mismatch[17].

6 Conclusions

We presented a novel two-layered approach for CHC model validation that relies on SMT proofs to provide additional correctness guarantees. The approach is

[15] See https://github.com/cvc5/cvc5/issues/9760.
[16] See https://github.com/ufmg-smite/carcara/issues/12.
[17] See https://github.com/cvc5/LFSC/issues/87.

Table 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 presented 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

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) evaluating the approach with other FOL theories, (ii) embedding the approach into CHC-based verification tooling, and (iii) designing 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.

Acknowledgements Rodrigo Otoni's work was supported by the Swiss National Science Foundation, via grant 200021_197353. Martin Blicha's work was supported by the Czech Science Foundation, via grant 23-06506 S. The authors thank Fedor Gromov for his assistance in preparing the SMT instance generator.

References

1. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity Compiler's Model Checker. In: Proceedings of the 34th International Conference on Computer Aided Verification. pp. 325–338 (2022)
2. Andreotti, B., Lachnitt, H., Barbosa, H.: 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. pp. 367–386 (2023)
3. Armand, M., Faure, G., Grégoire, B., Keller, C., Théry, L., Werner, B.: A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In: Proceedings of the 1st International Conference on Certified Programs and Proofs. pp. 135–150 (2011)
4. Baek, S., Carneiro, M., Heule, M.J.H.: 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. pp. 59–75 (2021)
5. 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., Zohar, Y.: 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. pp. 415–442 (2022)
6. Barbosa, H., Blanchette, J.C., Fleury, M., Fontaine, P.: Scalable Fine-Grained Proofs for Formula Processing. Journal of Automated Reasoning **64**(3), 485–510 (2020)
7. Barbosa, H., Hoenicke, J., Bobot, F.: SMT-COMP 2022: Competition Report. https://smt-comp.github.io/2022/slides-smtworkshop.pdf (2022)
8. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf (2021)
9. Barrett, C., Sebastiani, R., Seshia, S., Tinelli, C.: Satisfiability Modulo Theories. In: Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability. IOS Press (2021)
10. Beyer, D.: 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. pp. 495–522 (2023)
11. Beyer, D., Strejček, J.: Case Study on Verification-Witness Validators: Where We Are and Where We Go. In: Proceedings of the 29th International Symposium on Static Analysis. pp. 160–174 (2022)
12. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn Clause Solvers for Program Verification. Fields of Logic and Computation II. Lecture Notes in Computer Science **9300**, 24–51 (2015)
13. Blanchette, J.C., Böhme, S., Fleury, M., Smolka, S.J., Steckermeier, A.: Semi-intelligible Isar Proofs from Machine-Generated Proofs. Journal of Automated Reasoning **56**(2), 155–200 (2016)

14. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn Solver. In: Proceedings of the 35th International Conference on Computer Aided Verification (2023), [to be published]
15. Böhme, S., Weber, T.: Fast LCF-Style Proof Reconstruction for Z3. In: Proceedings of the 1st International Conference on Interactive Theorem Proving. pp. 179–194 (2010)
16. Bouton, T., Caminha B. de Oliveira, D., Déharbe, D., Fontaine, P.: veriT: An Open, Trustable and Efficient SMT-Solver. In: Proceedings of the 22nd International Conference on Automated Deduction. pp. 151–156 (2009)
17. Bury, G.: Dolmen: A Validator for SMT-LIB and Much More. In: Proceedings of the 19th International Workshop on Satisfiability Modulo Theories. pp. 32–39 (2021)
18. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and Sound Static Analysis of Android Applications by SMT Solving. In: Proceedings of the 1st IEEE European Symposium on Security and Privacy. pp. 47–62 (2016)
19. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Proceedings of the 19th International SPIN Workshop. pp. 248–254 (2012)
20. Cruz-Filipe, L., Heule, M.J.H., Hunt, W.A., Kaufmann, M., Schneider-Kamp, P.: Efficient Certified RAT Verification. In: Proceedings of the 26th International Conference on Automated Deduction. pp. 220–236 (2017)
21. De Angelis, E., Govind V K, H.: CHC-COMP 2022: Competition Report. In: Proceedings of the 9th Workshop on Horn Clauses for Verification and Synthesis. pp. 44–62 (2022)
22. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
23. Ekici, B., Mebsout, A., Tinelli, C., Keller, C., Katz, G., Reynolds, A., Barrett, C.: SMTCoq: A Plug-In for Integrating SMT Solvers into Coq. In: Proceedings of the 29th International Conference on Computer Aided Verification. pp. 126–133 (2017)
24. Ernst, G.: 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. pp. 559–564 (2023)
25. Fontaine, P., Marion, J.Y., Merz, S., Nieto, L.P., Tiu, A.: 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. pp. 167–181 (2006)
26. Gario, M., Micheli, A.: PySMT: a Solver-agnostic Library for Fast Prototyping of SMT-based Algorithms. In: Proceedings of the 13th International Workshop on Satisfiability Modulo Theories. pp. 1–10 (2015)
27. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing Software Verifiers from Proof Rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 405–416 (2012)
28. Gurfinkel, A., Bjørner, N.: The Science, Art, and Magic of Constrained Horn Clauses. In: Proceedings of the 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. p. 6–10 (2019)
29. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn Verification Framework. In: Proceedings of the 27th International Conference on Computer Aided Verification. pp. 343–361 (2015)

30. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Trimming While Checking Clausal Proofs. In: Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design. pp. 181–188 (2013)
31. Heule, M., Hunt, W., Kaufmann, M., Wetzler, N.: Efficient, Verified Checking of Propositional Proofs. In: Proceedings of the 8th International Conference on Interactive Theorem Proving. pp. 269–284 (2017)
32. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Verifying Refutations with Extended Resolution. In: Proceedings of the 24th International Conference on Automated Deduction. pp. 345–359 (2013)
33. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**(10), 576–580 (1969)
34. Hoenicke, J., Schindler, T.: A Simple Proof Format for SMT. In: Proceedings of the 20th International Workshop on Satisfiability Modulo Theories. pp. 54–70 (2022)
35. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn Clauses for Communicating Timed Systems. In: Proceedings of the 1st Workshop on Horn Clauses for Verification and Synthesis. pp. 39–52 (2014)
36. Hojjat, H., Rümmer, P.: The Eldarica Horn Solver. In: Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design. pp. 1–7 (2018)
37. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT Solver for Multi-core and Cloud Computing. In: Proceedings of the 19th International Conference on Theory and Applications of Satisfiability Testing. pp. 547–553 (2016)
38. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: A Framework for Verifying Java programs. In: Proceedings of the 28th International Conference on Computer Aided Verification. pp. 352–358 (2016)
39. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based Model Checking for Recursive Programs. Formal Methods in System Design **48**(3), 175–205 (2016)
40. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer Berlin, Heidelberg, 2 edn. (2016)
41. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A Verified Implementation of ML. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 179–191 (2014)
42. Lammich, P.: Efficient Verified (UN)SAT Certificate Checking. Journal of Automated Reasoning **64**(3), 513–532 (2020)
43. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-Based Verification for Rust Programs. ACM Transactions on Programming Languages and Systems **43**(4), 1–54 (2021)
44. de Moura, L., Bjørner, N.: Proofs and Refutations, and Z3. In: Proceedings of the 7th International Workshop on the Implementation of Logics. pp. 123–132 (2008)
45. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A.E.J., Sharygina, N.: Theory-Specific Proof Steps Witnessing Correctness of SMT Executions. In: Proceedings of the 58th ACM/IEEE Design Automation Conference. pp. 541–546 (2021)
46. Otoni, R., Marescotti, M., Alt, L., Eugster, P., Hyvärinen, A., Sharygina, N.: A Solicitous Approach to Smart Contract Verification. ACM Transactions on Privacy and Security **26**(2), 1–28 (2023)
47. Reeves, J.E., Kiesl-Reiter, B., Heule, M.J.H.: Propositional Proof Skeletons. In: Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 329–347 (2023)
48. Rümmer, P., Hojjat, H., Kuncak, V.: On Recursion-Free Horn Clauses and Craig Interpolation. Formal Methods In System Design **47**(1), 1–25 (2015)

49. Sandberg Ericsson, A., Myreen, M.O., Åman Pohjola, J.: A Verified Generational Garbage Collector for CakeML. In: Proceedings of the 8th International Conference on Interactive Theorem Proving. pp. 444–461 (2017)
50. Schurr, H.J., Fleury, M., Barbosa, H., Fontaine, P.: Alethe: Towards a Generic SMT Proof Format. In: Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving. pp. 49–54 (2021)
51. Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Proceedings of the 1st International Symposium on Computer Science in Russia. pp. 600–611 (2006)
52. Stump, A., Oe, D., Reynolds, A., Hadarean, L., Tinelli, C.: SMT Proof Checking Using a Logical Framework. Formal Methods in System Design **42**(1), 91–118 (2013)
53. Tange, O.: GNU Parallel - The Command-Line Power Tool. ;login: The USENIX Magazine **36**(1), 42–47 (2011)
54. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In: Proceedings of the 17th International Conference on Theory and Applications of Satisfiability Testing. pp. 422–429 (2014)