Validation of CHC Satisfiability with ATHENA

RODRIGO OTONI, USI Lugano, Lugano, Switzerland
MARTIN BLICHA, USI Lugano, Lugano, Switzerland and Charles University, Prague, Czech Republic
PATRICK EUGSTER, USI Lugano, Lugano, Switzerland
NATASHA SHARYGINA, USI Lugano, Lugano, Switzerland

Formal verification tooling increasingly relies on logic solvers as automated reasoning engines. A 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 it 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 effort in result validation. We propose a two-layered validation approach for witnesses of CHC satisfiability that validates CHC models via proof-backed SMT queries. We developed a modular evaluation framework, ATHENA, and assessed the approach's viability via large scale experimentation, comparing three CHC solvers, five SMT solvers, and five proof checkers. Our results indicate that the approach is feasible, with the potential to be incorporated into CHC-based tooling, and also confirm the need for validation, with fourteen bugs being found in the tools used.

CCS Concepts: • **Theory of computation** → **Automated reasoning**; **Logic and verification;**

Additional Key Words and Phrases: Validation, constrained Horn clauses, SMT proofs

ACM Reference Format:
Rodrigo Otoni, Martin Blicha, Patrick Eugster, and Natasha Sharygina. 2025. Validation of CHC Satisfiability with ATHENA. *Form. Asp. Comput.* 37, 4, Article 30 (October 2025), 20 pages. https://doi.org/10.1145/3716505

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) [32]. The CHC fragment has been shown to be a match for Hoare logic [38] with practical uses [11], aiding in reasoning about the behavior

Rodrigo Otoni's work was supported by the Swiss National Science Foundation grant 200021_197353. Martin Blicha's work was supported by the Swiss National Science Foundation grant 200021_185031 and the Czech Science Foundation grant 23-06506S.
Authors' Contact Information: Rodrigo Otoni, USI Lugano, Lugano, Switzerland; e-mail: otonir@usi.ch; Martin Blicha, USI Lugano, Lugano, Switzerland and Charles University, Prague, Czech Republic; e-mail: blichm@usi.ch; Patrick Eugster, USI Lugano, Lugano, Switzerland; e-mail: eugstp@usi.ch; Natasha Sharygina, USI Lugano, Lugano, Switzerland; e-mail: sharygin@usi.ch.

ACM 1433-299X/2025/10-ART30
https://doi.org/10.1145/3716505

of procedural [11] and functional [31] programs, as well as concurrent systems [40] and smart contracts [50], 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 [44], 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 [41], Golem [13], and Spacer [43], serving for instance, as back-end reasoning engines of verification tools targeting programs written in C/C++ [33], Java [42], Rust [47], and Solidity [1], as well as Android applications [19].

1.1 Need for Validation

Despite their extensive usage in verification, logic solvers are themselves not immune to bugs. To illustrate this point, in the 2023 edition of the annual SMT competition, there were 300 benchmarks in which at least two state-of-the-art solvers disagreed on the results, with in total 10 competing solvers producing unsound results [16]. 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) [45] and garbage collector [54] 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 from being 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, 21, 35, 37, 56, 60] and SMT [22, 39, 48, 55, 57] solvers, with codebase verification being applied instead to the validation tools [36, 46], which are much less complex and easier to maintain. Both the annual SAT and SMT competitions now incorporate witness usage in some capacity[1] and the organizers of the annual CHC competition, CHC-COMP, have the validation of witnesses as a goal for future editions [27].

1.2 CHC Witnesses

The input of a CHC solver, detailed in Section 3, 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 a UNSAT result, called *UNSAT proof*, should contain an explanation of how *false* can be derived, while a witness for an SAT result, called *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 article, UNSAT proofs and SAT models will be referred to simply as *proofs* and *models*.

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 [23].[2] Unlike the case for models, however, the proofs produced cannot be independently validated with available tooling, given that, to the best of our knowledge, no proof-checking approach currently exists.

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

1.3 Two-Layered Model Validation

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.

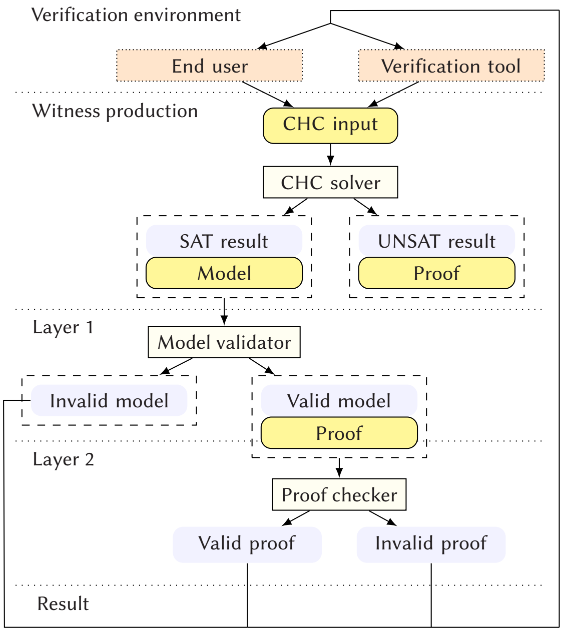


Fig. 1. Overview of our two-layered approach for validation of CHC satisfiability. CHC models are validated through proof-backed SMT queries. Although capable of being produced, CHC proofs cannot be checked currently.

1.4 The ATHENA Framework

To assess 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 used it to conduct a large scale evaluation. Concretely, we used our framework to validate the models produced by three CHC solvers, ELDARICA [41], GOLEM [13], and SPACER [43], with each model produced being separately validated, in Layer 1, by five proof producing SMT solvers, cvc5 [5], OPENSMT [17], SMTINTERPOL [20], VERIT [15], and Z3 [23]. 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 ALFC [52], CARCARA [2], LFSC checker [57], SMTINTERPOL checker [39], and TSWC [48].

1.5 Contributions

In addition to the detailed description of our validation approach and the ATHENA framework, we also report the results of an extensive evaluation targeting two FOL theories, namely the linear integer arithmetic (LIA) theory and the theory of arrays combined with LIA (ALIA). We used all LIA and ALIA benchmarks from CHC-COMP 2024 in our evaluation, 450 containing only linear Horn clauses, i.e., implications with a single uninterpreted predicate in the implicant, and 600 containing nonlinear Horn clauses, i.e., implications with multiple uninterpreted predicates in the implicant. The benchmarks used led to 63,994 SMT instances and 299,658 SMT proofs being produced as part of the validation process.

Three observations can be made from the results obtained. First, the proposed proof-backed model validation approach 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 [33] and SOLCMC [1] model checkers, can in principle benefit from the guarantees provided by model validation. In particular, invariant generation in CHC-based tools, which commonly rely on models, can be

Table 1. Brief Descriptions of the Bugs Found During the Evaluation

		Bug effect
CHC solvers	Eldarica	Invalid model produced[9]
	Eldarica	Crash due to quantifiers in predicates[6]
	Spacer	Invalid model produced[11]
	Golem	Syntactically malformed model produced (x2)[8]
	Golem	Crash during model production[7]
SMT solvers	cvc5	Invalid proof produced[15]
	cvc5	Crash during proof production (x2)[13]
	OpenSMT	Crash during sort inference[12]
	SMTInterpol	Invalid proof produced[14]
Proof checkers	alfc	Crash during proof checking[18]
	Carcara	Parsing error due to unknown attribute[16]
	LFSC checker	Crash during type inference[17]

made more robust. Second, model and proof sizes, which reached the order of gigabytes in our experiments, 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 [51], 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 three SMT solvers and three 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 6. 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 approach to validate CHC satisfiability results.
(2) Development of a modular evaluation framework, ATHENA, to make the approach practical.
(3) Staging of a large scale evaluation to determine the viability of our approach.

An earlier version of this work was published as a conference paper [49]. The present article substantially extends and improves the previous one, in the following fivefold manner:

(I) Addition of support for the theory of arrays with linear integer arithmetic to ATHENA, which demonstrates our approach's generality in regards to FOL theories.
(II) Integration to our framework of (i) a postprocessing step for aggregation of layers' results per input file and of (ii) a new SMT proof format, AletheLF, and of its checker, alfc, which together provide more information and validation options to the user.
(III) Extension of our evaluation to include 1050 new benchmarks, timeouts of 30 minutes, and the latest stable versions of thirteen state-of-the-art tools, which combined provide an extremely detailed picture of CHC satisfiability validation in practice.
(IV) Reporting of five new bugs uncovered during our extended evaluation, including the production of invalid proofs by SMTInterpol.
(V) Improvement of explanations in many parts of the manuscript, including the addition of an algorithmic description of our validation approach (in Section 4).

1.6 Roadmap

The remainder of the article is structured as follows. Related witness validation approaches are covered in Section 2. The necessary CHC background is given in Section 3. The two layers of our validation approach are presented in Section 4. ATHENA is detailed in Section 5. The evaluation is discussed in Section 6. Finally, our conclusions are laid out in Section 7.

2 Related Work

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 this section we provide an overview of related witness validation approaches, targeting SAT and SMT solving as well as other contexts. For details on SAT and SMT solving we refer readers to the book of Kroening and Strichman [44].

2.1 Validation of SAT Witnesses

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 tradeoffs between proof compactness and checking efficiency. Initial formats were based on resolution [56] and clausal proofs [35], with resolution asymmetric tautology (RAT) [37] being a base for many recent developments, e.g., deletion RAT (DRAT) [60], linear RAT (LRAT) [21], 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 [60] being the standard proof checker for proofs following this format.

2.2 Validation of SMT Witnesses

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 organizers of the SMT competition having been made in recent years for the adoption of a unified format in line with the SMT-LIB standard [7]. A separate, experimental, model validation track has been established and has seen a steady increase in the SMT-LIB logics supported, with PySMT [30] and Dolmen [18] 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 [55] was initially supported by veriT, but has since also been integrated into cvc5's proof production. cvc5 also caters for proofs based on the logical framework with side conditions (LFSC) [57], with LFSC support preceding Alethe's integration and dating back to the CVC3 version of the tool. SMTInterpol [39], OpenSMT [48], and Z3 [22] 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. On 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, 25] and Isabelle/HOL [6, 12, 14, 28]. When it comes to automated checkers, their goal is mainly to serve as independent lightweight validators, with the potential to be integrated into tools such as model checkers. Automated checkers are available in a variety of formats [2, 39, 48, 57, 60]. 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 existing formats.

2.3 Validation of Other Types of Witnesses

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 [9]. Software verification witnesses are different from those used by logic solvers, being categorized as either correctness or violation witnesses, with their own formats[3] and limitations [10]. The tool that may best illustrate the usage of witnesses is Korn [26], 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 Constrained Horn Clauses

The constrained Horn clauses formalism has been proposed as a unified, purely logic-based, intermediate language for reasoning about verification tasks [31]. 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.

Following standard SMT terminology [8], 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.* Furthermore, a clause is usually called a *query* if its head is equal to *false* and a *fact* if no uninterpreted predicates are present in its body.

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 evaluates 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* [53]. Having LIA as our theory $\mathcal{T}$, an example of an interpretation is that of a set of (in)equalities, with the elements of the set being conjoined when replacing the predicate they are an interpretation for. 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.

4 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 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

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

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 3, 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 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.

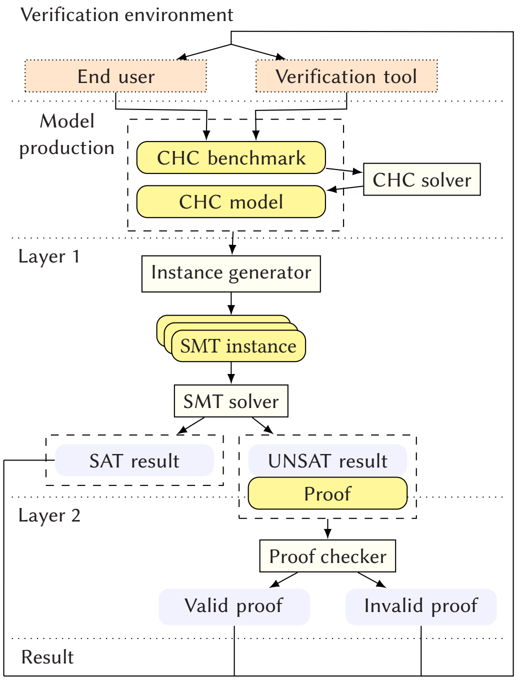


Fig. 2. Breakdown of our two-layered approach for validation of CHC satisfiability. A valid model will have all the SMT instances generated from it yield an UNSAT result backed by a valid SMT proof.

As an example of the computation done in the first layer of our approach, consider the following CHC system, consisting of three Horn clauses and a single uninterpreted predicate *Inv*:

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

This system is satisfiable with a potential model being one that contains the interpretation $Inv(x) \equiv x \leq 5$. 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 all unsatisfiable in the LIA theory, which can be trivially seen for this small example as the assignments to x in each formula lead to a contradiction:

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

ALGORITHM 1: Validation of a CHC model.

```
input  : pair $(\mathcal{S}, \mathcal{M})$, with $\mathcal{S}$ being a set of CHC clauses and $\mathcal{M}$ being a candidate model
output : Boolean value representing whether $\mathcal{M}$ is a valid model for $\mathcal{S}$ or not
1 foreach clause $C \in \mathcal{S}$ do
2     $C_I \leftarrow$ InstantiateClause($C, \mathcal{M}$)
3     $(answer, proof) \leftarrow$ CheckSMTSatisfiability($\neg C_I$)
4     if $answer \neq$ UNSAT then return false
5     if not ValidateSMTProof($\neg C_I, proof$) then return false
6 return true
```

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; a generic example of the computation done in the second layer is not possible because it is specific to the proof format and proof checker being used.

The pseudocode describing the whole validation process can be seen in Algorithm 1. The `InstantiateClause` function replaces the uninterpreted predicates in the given clause by the interpretations present in the model, while the `CheckSMTSatisfiability` and `ValidateSMTProof` functions stand for calls to an SMT solver and a proof checker. 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.

5 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 3,485 lines of Shell and Python code in total, is fully automated, and uses GNU PARALLEL [58] to achieve a large degree of parallelization in order to better tackle the high computation cost. One important assumption is the correctness of the instance generator, i.e., the `InstantiateClause` function in Algorithm 1. The generator's small size, 197 lines of Python code, makes manual inspection a reasonable way to ensure correctness at this stage. ATHENA is open-source,[4] 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.

6 Evaluation

We first describe the benchmarks and tools used, in Section 6.1, and then discuss the results obtained related to CHC model validation, i.e., layer 1, in Section 6.2, and SMT proof checking,

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

i.e., layer 2, in Section 6.3. 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 30 minutes and a memory limit of 10 gigabytes. In total, our evaluation took 2,392 hours of CPU time.

6.1 Benchmarks and Tools

We used all benchmarks of the two LIA and the two LIA-Arrays tracks of CHC-COMP 2024 [27]. The LIA-lin and LIA-Arrays-lin tracks consist of benchmarks containing only linear Horn clauses, and the LIA-nonlin and LIA-Arrays-nonlin tracks consist 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. Regarding the LIA-Arrays benchmarks, their tracks are also very traditional, being present in the competition since its second edition and also having a significant number of competitors, and the ALIA theory is covered by three out of five of the available proof producing SMT solvers.

For model production, we chose the three best performing CHC solvers in the LIA tracks of CHC-COMP 2023 for comparison, which are, in alphabetical order, Eldarica [41], Golem [13], and Spacer [43]. Regarding the other competitors, LoAT [29] and Theta [59] are currently not capable of producing models, and the two competitors based on the Ultimate framework, Ultimate TreeAutomizer [24] and Ultimate Unihorn [34], have no clear build instructions available.

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 [17], SMTInterpol [20], and veriT [15], as well as Z3 [23], which can produce proofs but did not compete in the track. To check the SMT proofs we used the fully automated checkers alfc [52], for AletheLF proofs produced by cvc5, Carcara [2], for Alethe proofs produced by cvc5 and veriT, LFSC checker [57], for LFSC proofs produced by cvc5, SMTInterpol checker [39], for proofs produced by SMTInterpol, and TSWC [48], for proofs produced by OpenSMT; to the best of our knowledge there is currently no independent automated checker for proofs produced by Z3.[5]

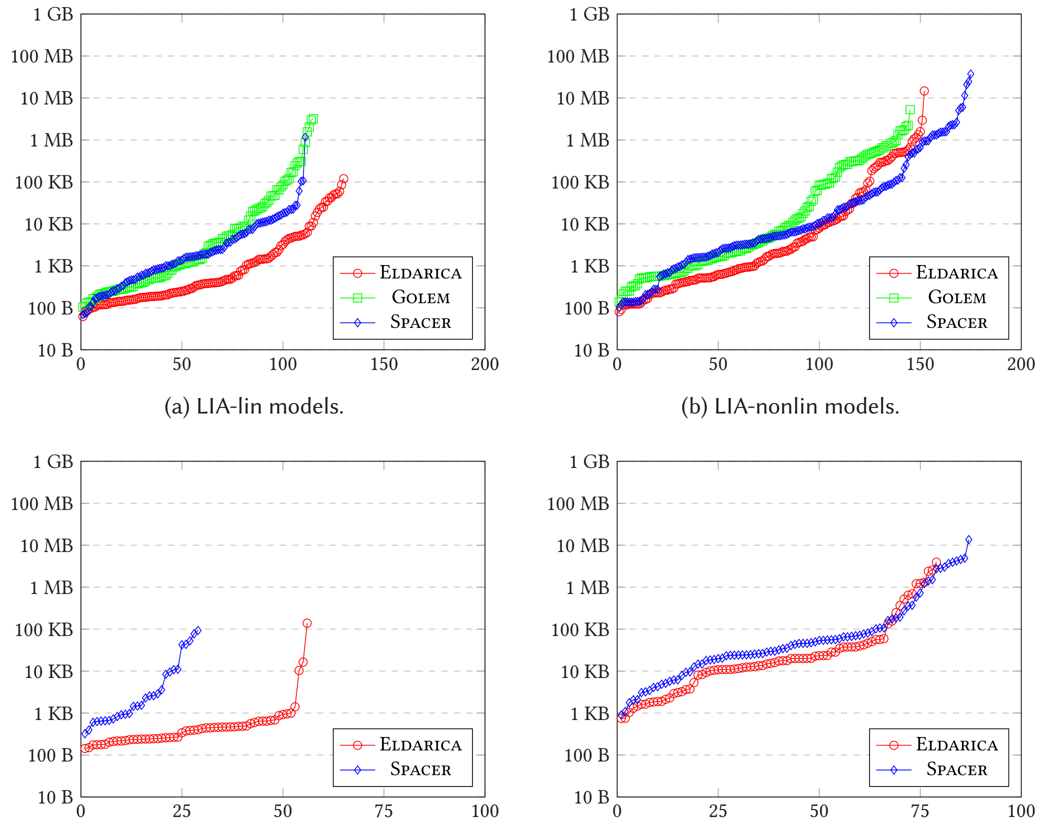
6.2 Model Validation Results

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 their default engine configurations, with Golem currently not supporting benchmarks containing arrays. The performance of the tools is in line with the CHC-COMP results, with 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, being yielded only by Spacer in four of its executions. Three errors, i.e., tool crashes, were observed with Eldarica, due to the presence of quantifiers in predicate interpretations computed.[6] A number of crashes were also observed with Golem while testing our framework,[7] as well as syntactically malformed models being produced by it,[8] with the underlying causes of both issues being addressed before the full-scale evaluation. Regarding the models' sizes (in bytes), Eldarica's models tended to be the most compact, followed by Spacer's, with Golem producing most of the larger models, as can be seen in Figure 3. The last point of note is that all models produced by Golem are quantifier-free, while Eldarica produced 68 quantified model, 7 from LIA-Arrays-lin

[5]For details on proof efforts involving the Z3 solver see https://microsoft.github.io/z3guide/programming/Proof Logs.
[6]See https://github.com/uuverifiers/eldarica/issues/46 and ../eldarica/issues/39.
[7]See https://github.com/usi-verification-and-security/golem/issues/29.
[8]See https://github.com/usi-verification-and-security/golem/issues/27 and ../golem/issues/68.

Table 2. Results for Solving the CHC Benchmarks of the LIA and LIA-Arrays Tracks

		SAT	UNSAT	Unknown	Timeout	Memout	Error
LIA-lin (300 benchmarks)	Eldarica	130	59	0	110	1	0
	Golem	115	67	0	118	0	0
	Spacer	111	60	2	88	39	0
LIA-nonlin (300 benchmarks)	Eldarica	152	92	0	55	1	0
	Golem	145	98	0	57	0	0
	Spacer	175	95	2	25	3	0
LIA-Arrays-lin (150 benchmarks)	Eldarica	56	23	0	70	0	1
	Golem	-	-	-	-	-	-
	Spacer	29	27	0	94	0	0
LIA-Arrays-nonlin (300 benchmarks)	Eldarica	79	66	0	153	0	2
	Golem	-	-	-	-	-	-
	Spacer	87	109	0	98	6	0



(a) LIA-lin models.

(b) LIA-nonlin models.

(c) LIA-Arrays-lin models.

(d) LIA-Arrays-nonlin models.

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.

benchamrks and 61 from LIA-Arrays-nonlin benchmarks, and Spacer produced 238 quantified models, 42 from LIA-lin benchmarks, 97 from LIA-nonlin benchmarks, 19 from LIA-Arrays-lin benchmarks, and 80 from LIA-Arrays-nonlin benchmarks.

Table 3. Results for Solving the SMT Instances Generated from the LIA Models

		SAT	UNSAT	Unknown	Timeout	Memout	Error	Unsupported
LIA-lin ELDARICA (1,433 instances)	CVC5	0	1,426	0	1	0	6	0
	OPENSMT	0	1,427	0	0	0	6	0
	SMTINTERPOL	0	1,427	0	0	0	6	0
	VERIT	0	1,348	0	0	0	6	79
	Z3	0	1,433	0	0	0	0	0
LIA-lin GOLEM (1,004 instances)	CVC5	0	1,004	0	0	0	0	0
	OPENSMT	0	1,004	0	0	0	0	0
	SMTINTERPOL	0	1,004	0	0	0	0	0
	VERIT	0	964	0	1	0	0	39
	Z3	0	1,004	0	0	0	0	0
LIA-lin SPACER (1,273 instances)	CVC5	4	1,269	0	0	0	0	0
	OPENSMT	0	647	0	0	0	0	626
	SMTINTERPOL	4	1,268	1	0	0	0	0
	VERIT	0	602	0	0	0	0	671
	Z3	4	1,269	0	0	0	0	0
LIA-nonlin ELDARICA (13,540 instances)	CVC5	0	13,529	0	0	0	11	0
	OPENSMT	0	13,529	0	0	0	11	0
	SMTINTERPOL	0	13,529	0	0	0	11	0
	VERIT	0	13,463	0	0	0	11	66
	Z3	1	13,539	0	0	0	0	0
LIA-nonlin GOLEM (13,109 instances)	CVC5	0	13,109	0	0	0	0	0
	OPENSMT	0	13,109	0	0	0	0	0
	SMTINTERPOL	0	13,106	0	3	0	0	0
	VERIT	0	13,102	0	0	7	0	0
	Z3	0	13,105	0	0	4	0	0
LIA-nonlin SPACER (14,716 instances)	CVC5	39	14,677	0	0	0	0	0
	OPENSMT	3	5,028	0	0	0	0	9,685
	SMTINTERPOL	39	14,297	146	199	35	0	0
	VERIT	3	5,027	0	0	0	0	9,686
	Z3	39	14,675	0	2	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 and LIA-Arrays tracks can be seen in Tables 3 and 4, respectively; the number of SMT instances generated for each CHC solver is related to the number of models it produced, with VERIT currently not supporting instances containing arrays. The combination of related tools, e.g., pairing of SPACER with Z3 or GOLEM with OPENSMT, might yield better solving performance, but it weakens the strength of independent validation.

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 from the SMT solver, was observed. Both ELDARICA and SPACER produced invalid models, although the latter to a significantly higher degree. ELDARICA's invalid model is due to the

Table 4. Results for Solving the SMT Instances Generated from the LIA-Arrays Models

		SAT	UNSAT	Unknown	Timeout	Memout	Error	Unsupported
LIA-Arrays-lin Eldarica (880 instances)	cvc5	0	880	0	0	0	0	0
	OpenSMT	0	844	0	0	0	0	36
	SMTInterpol	0	880	0	0	0	0	0
	veriT	-	-	-	-	-	-	-
	Z3	0	880	0	0	0	0	0
LIA-Arrays-lin Spacer (365 instances)	cvc5	0	356	5	4	0	0	0
	OpenSMT	0	140	0	0	0	0	225
	SMTInterpol	0	355	5	5	0	0	0
	veriT	-	-	-	-	-	-	-
	Z3	0	363	2	0	0	0	0
LIA-Arrays-nonlin Eldarica (8,348 instances)	cvc5	0	5,348	106	0	0	2,894	0
	OpenSMT	0	311	0	0	0	169	7,868
	SMTInterpol	0	5,305	86	86	10	2,861	0
	veriT	-	-	-	-	-	-	-
	Z3	0	5,407	31	16	9	2,885	0
LIA-Arrays-nonlin Spacer (9,326 instances)	cvc5	17	8,577	710	20	1	0	0
	OpenSMT	0	189	0	0	0	0	9,137
	SMTInterpol	5	8,416	259	638	8	0	0
	veriT	-	-	-	-	-	-	-
	Z3	16	8,688	583	39	0	0	0

embedding of Boolean values into arithmetic operations,[9] which leads to an error in most SMT solvers and is the cause of the errors reported in the tables, but can be solved by Z3 in some cases via unit propagation[10]. Spacer's invalid models are due to problematic internal transformations.[11] While not implying that the SAT results the models are supposed to serve as witnesses for are incorrect, since the problem can be in the model production itself, this is a serious issue. 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 occurrence of errors when OpenSMT was handling instances generated from Eldarica and Spacer models in our preliminary experiments, which was due to a limitation in scoping in the presence of different sorts[12] that was fixed prior to the full-scale evaluation, and the small, but consistent, number of instances unsupported solely 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 first-order logic, use operators reserved for the nonlinear integer arithmetic (NIA) logic of the SMT-LIB standard; the competition organizers were informed of this finding.

6.3 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. In addition to veriT not handling arrays, cvc5-Alethe and OpenSMT cannot produce proofs for instances containing arrays. Since proof production adds an overhead to solver execution, the

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

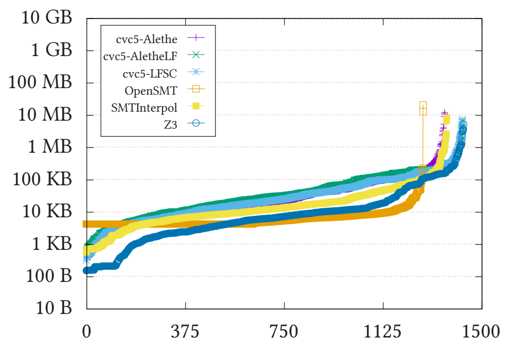
	Proofs produced					
	LIA-lin			LIA-nonlin		
	Eldarica (1,433)	Golem (1,004)	Spacer (1,273)	Eldarica (13,540)	Golem (13,109)	Spacer (14,716)
cvc5-Alethe	1,360	865	1,029	12,941	12,039	9,473
cvc5-AletheLF	1,427	1,000	1,269	13,528	13,106	14,675
cvc5-LFSC	1,427	1,002	1,269	13,528	13,109	14,676
OpenSMT	1,279	1,004	647	12,208	13,109	1,336
SMTInterpol	1,368	935	1,199	13,244	12,827	13,736
veriT	0	0	0	0	0	0
Z3	1,432	1,004	1,269	13,539	13,104	14,675
	LIA-Arrays-lin			LIA-Arrays-nonlin		
	Eldarica (880)	Golem -	Spacer (365)	Eldarica (8,348)	Golem -	Spacer (9,326)
cvc5-Alethe	-	-	-	-	-	-
cvc5-AletheLF	880	-	344	3,414	-	5,427
cvc5-LFSC	880	-	345	4,709	-	7,943
OpenSMT	-	-	-	-	-	-
SMTInterpol	880	-	353	5,216	-	8,294
veriT	-	-	-	-	-	-
Z3	880	-	364	5,405	-	8,694

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.

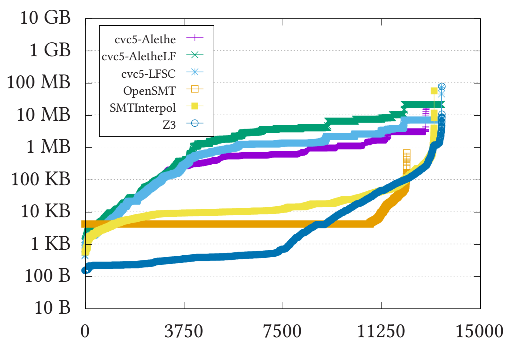
number of proofs produced is expected to be lower than the amount of UNSAT results reported in Tables 3 and 4. Concretely, the combined percentage of proofs produced in relation to the previous UNSAT results, for the ten instance sets (or six if arrays are not supported), is: 83% for cvc5-Alethe, 91% for cvc5-AletheLF, 97% for cvc5-LFSC, 85% for OpenSMT, 97% for SMTInterpol, 0% for veriT, and 100% for Z3. The reduction in performance observed is overall small for all solvers, with the main reason for the lower amount of UNSAT results in proof production mode being the errors present exclusively in this mode. The production of proofs by cvc5, particularly of Alethe proofs, is currently unstable[13], while OpenSMT in proof production mode still suffers from scoping problems[12]. One note is that veriT was not able to produce any proofs, with the reason being 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.

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 Figures 4 and 5. The ranking of proof sizes between the SMT solvers 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. The single largest proof produced, by cvc5-LFSC from an instance generated

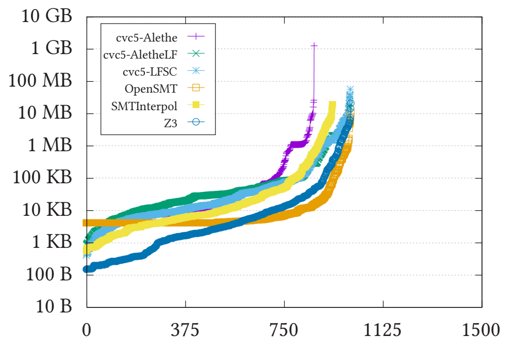
[13]See https://github.com/cvc5/cvc5/issues/9770 and ../cvc5/issues/10836.



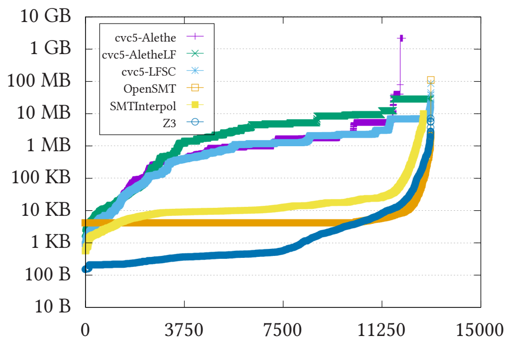
(a) LIA-lin proofs for Eldarica models.



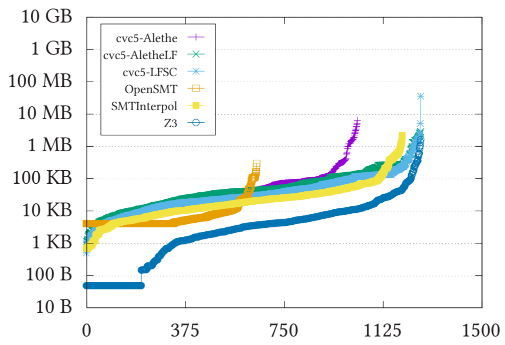
(b) LIA-nonlin proofs for Eldarica models.



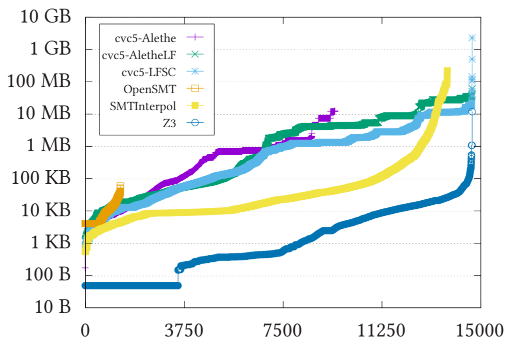
(c) LIA-lin proofs for Golem models.



(d) LIA-nonlin proofs for Golem models.



(e) LIA-lin proofs for Spacer models.

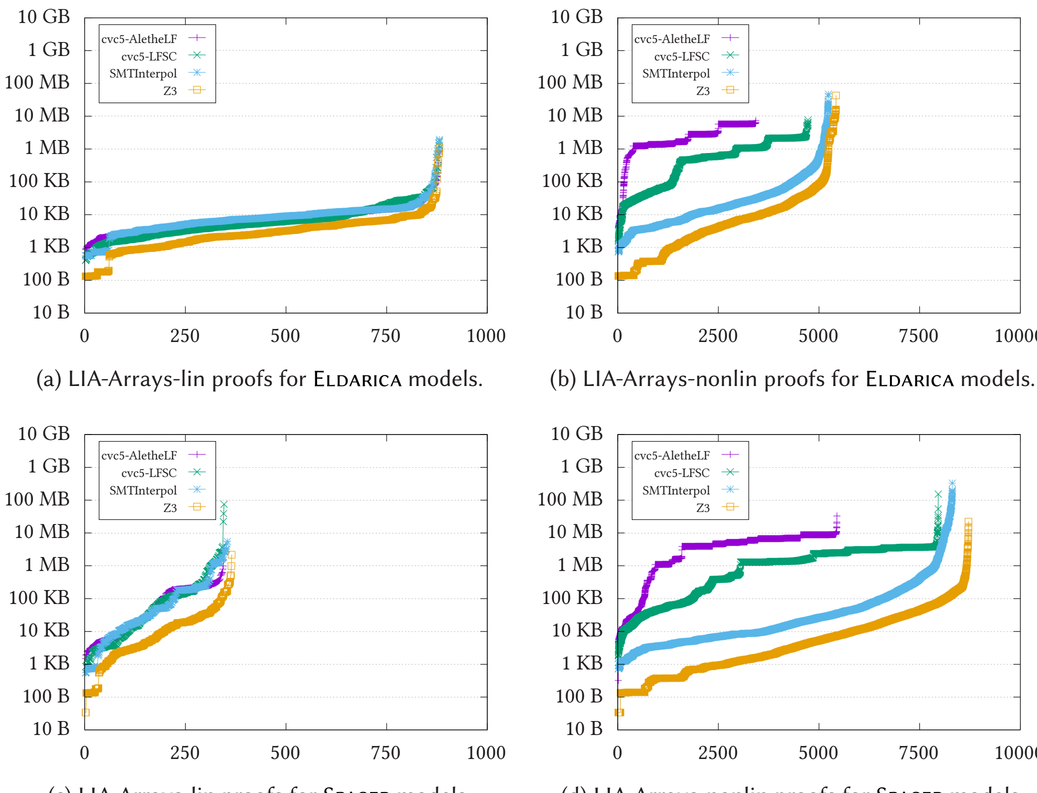


(f) LIA-nonlin proofs for Spacer models.

Fig. 4. Sizes of the LIA 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.

from a Spacer LIA-nonlin model, had a size of 2.3 gigabytes, which is a good illustration of the need of compact proofs.

To check the proofs we used the available automated checkers suitable for each proof format, namely alfc, Carcara, and TSWC for the proofs produced by cvc5-AletheLF, 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 instance sets of the LIA and LIA-Arrays tracks can be seen in Tables 6 and 7, respectively. Overall, the five checkers were able to



(a) LIA-Arrays-lin proofs for Eldarica models. (b) LIA-Arrays-nonlin proofs for Eldarica models.

(c) LIA-Arrays-lin proofs for Spacer models. (d) LIA-Arrays-nonlin proofs for Spacer models.

Fig. 5. Sizes of the LIA-Arrays 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.

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 10 gigabytes. An important discovery is that SMTInterpol produced 25 invalid proofs[14]. 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. This is combined with the 562 invalid proofs produced by cvc5-Alethe, due to incorrect proof steps[15], uncovered in our iFM'23 experiments [49]. Still in regards to our previous experiments, they also led to 44,282 errors with Carcara, when checking proofs produced for SMT instances generated from Spacer models due to the presence of attribute annotations in models containing quantifiers[16], and three errors with the LFSC checker, due to a type mismatch[17]. Lastly, 27 errors were also found with the newly integrated checker alfc[18].

With the production of a CHC model, the generation of the associated SMT instances, the solving of the instances and the production of SMT proofs, and finally the checking of the proofs, it is possible to say with a high degree of confidence that a CHC benchmark is satisfiable. Applying this whole process to our benchmarks, we obtain the results shown in Table 8.

[14]See https://github.com/ultimate-pa/smtinterpol/issues/148.
[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.
[18]See https://github.com/cvc5/alfc/issues/49.

Table 6. Results for Checking the Proofs Produced by Solving the SMT Instances Generated for the LIA Benchmarks

		Valid	Invalid	Timeout	Memout	Error
LIA-lin Eldarica	alfc	1,427 (100%)	0	0	0	0
	Carcara	1,360 (100%)	0	0	0	0
	LFSC	1,427 (100%)	0	0	0	0
	SMTInterpol	1,368 (100%)	0	0	0	0
	TSWC	1,279 (100%)	0	0	0	0
LIA-lin Golem	alfc	995 (99.5%)	0	1	4	0
	Carcara	865 (100%)	0	0	0	0
	LFSC	996 (99.4%)	0	2	4	0
	SMTInterpol	935 (100%)	0	0	0	0
	TSWC	1,004 (100%)	0	0	0	0
LIA-lin Spacer	alfc	1,269 (100%)	0	0	0	0
	Carcara	1,029 (100%)	0	0	0	0
	LFSC	1,269 (100%)	0	0	0	0
	SMTInterpol	1,199 (100%)	0	0	0	0
	TSWC	647 (100%)	0	0	0	0
LIA-nonlin Eldarica	alfc	13,527 (99.9%)	0	0	1	0
	Carcara	12,941 (100%)	0	0	0	0
	LFSC	13,517 (99.9%)	0	0	11	0
	SMTInterpol	13,244 (100%)	0	0	0	0
	TSWC	12,208 (100%)	0	0	0	0
LIA-nonlin Golem	alfc	13,106 (100%)	0	0	0	0
	Carcara	11,950 (99.2%)	0	0	89	0
	LFSC	13,097 (99.9%)	0	0	12	0
	SMTInterpol	12,827 (100%)	0	0	0	0
	TSWC	13,108 (99.9%)	0	1	0	0
LIA-nonlin Spacer	alfc	14,658 (99.8%)	0	0	0	17
	Carcara	9,472 (99.9%)	0	0	1	0
	LFSC	14,666 (99.9%)	0	1	9	0
	SMTInterpol	13,731 (99.9%)	2	3	0	0
	TSWC	1,336 (100%)	0	0	0	0

The percentage of proofs validated in relation to their total amount is shown in parentheses.

7 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 supported by a modular evaluation framework, ATHENA, that allows 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 and ALIA benchmarks from CHC-COMP 2024 to compare three CHC solvers, five SMT solvers, and five proof checkers. The results indicate that the approach is feasible in practice, with potential to benefit CHC-based verification tools; the results also highlight 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.

Table 7. Results for Checking the Proofs Produced by Solving the SMT Instances Generated for the LIA-Arrays Benchmarks

		Valid	Invalid	Timeout	Memout	Error
LIA-Arrays-lin Eldarica	alfc	880 (100%)	0	0	0	0
	Carcara	-	-	-	-	-
	LFSC checker	880 (100%)	0	0	0	0
	SMTInterpol	880 (100%)	0	0	0	0
	TSWC	-	-	-	-	-
LIA-Arrays-lin Spacer	alfc	343 (99.7%)	0	1	0	0
	Carcara	-	-	-	-	-
	LFSC checker	345 (100%)	0	0	0	0
	SMTInterpol checker	353 (100%)	0	0	0	0
	TSWC	-	-	-	-	-
LIA-Arrays-nonlin Eldarica	alfc	3,414 (100%)	0	0	0	0
	Carcara	-	-	-	-	-
	LFSC checker	4,695 (99.7%)	0	0	14	0
	SMTInterpol checker	5,193 (99.5%)	23	0	0	0
	TSWC	-	-	-	-	-
LIA-Arrays-nonlin Spacer	alfc	5,416 (99.7%)	0	1	0	10
	Carcara	-	-	-	-	-
	LFSC checker	6,081 (76.5%)	0	0	1,862	0
	SMTInterpol checker	8,292 (99.9%)	0	2	0	0
	TSWC	-	-	-	-	-

The percentage of proofs validated in relation to their total amount is shown in parentheses.

Table 8. Number of Benchmarks Shown to be Satisfiable by Our Two-layered Approach

	CHC benchmarks validated					
	LIA-lin			LIA-nonlin		
	Eldarica (130)	Golem (115)	Spacer (111)	Eldarica (152)	Golem (145)	Spacer (175)
cvc5-Alethe	84	63	48	99	64	20
cvc5-AletheLF	128	107	109	150	142	161
cvc5-LFSC	128	109	110	144	135	163
OpenSMT	105	115	69	112	144	48
SMTInterpol	113	99	93	129	119	117
veriT	0	0	0	0	0	0
Z3	-	-	-	-	-	-
	LIA-Arrays-lin			LIA-Arrays-nonlin		
	Eldarica (56)	Golem -	Spacer (29)	Eldarica (79)	Golem -	Spacer (87)
cvc5-Alethe	-	-	-	-	-	-
cvc5-AletheLF	56	-	24	12	-	23
cvc5-LFSC	56	-	24	16	-	29
OpenSMT	-	-	-	-	-	-
SMTInterpol	56	-	25	13	-	12
veriT	-	-	-	-	-	-
Z3	-	-	-	-	-	-

Each column shows the amount of benchmarks shown to be satisfiable by a combination of CHC solver, SMT solver, and proof checker, with the total amount of benchmarks to be validated by each CHC solver displayed in parentheses.

Directions for future work are threefold, namely, (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 those already supported, with algebraic data types being a suitable candidate for this. 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.

Acknowledgments

The authors thank Fedor Gromov for his help in preparing the instance generator.

References

[1] Leonardo Alt, Martin Blicha, Antti E. J. Hyvärinen, and Natasha Sharygina. 2022. SolCMC: Solidity compiler's model checker. In *Proceedings of the 34th International Conference on Computer Aided Verification*. 325–338.

[2] Bruno Andreotti, Hanna Lachnitt, and Haniel Barbosa. 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*. 367–386.

[3] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. 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*. 135–150.

[4] Seulkee Baek, Mario Carneiro, and Marijn J. H. Heule. 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*. 59–75.

[5] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. 2022. 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*. 415–442.

[6] Haniel Barbosa, Jasmin Christian Blanchette, Mathias Fleury, and Pascal Fontaine. 2020. Scalable fine-grained proofs for formula processing. *Journal of Automated Reasoning* 64, 3 (2020), 485–510.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2021. The SMT-LIB Standard: Version 2.6. Retrieved August 16, 2024 from https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf

[8] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. 2021. Satisfiability modulo theories. In *Handbook of Satisfiability, Second Edition, of Frontiers in Artificial Intelligence and Applications*, Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh (Eds.). Vol. 336, 825–885.

[9] Dirk Beyer. 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*. 495–522.

[10] Dirk Beyer and Jan Strejček. 2022. Case study on verification-witness validators: Where we are and where we go. In *Proceedings of the 29th International Symposium on Static Analysis*. 160–174.

[11] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. *Fields of Logic and Computation II. Lecture Notes in Computer Science* 9300 (2015), 24–51.

[12] Jasmin Christian Blanchette, Sascha Böhme, Mathias Fleury, Steffen Juilf Smolka, and Albert Steckermeier. 2016. Semi-intelligible isar proofs from machine-generated proofs. *Journal of Automated Reasoning* 56, 2 (2016), 155–200.

[13] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. 2023. The golem horn solver. In *Proceedings of the 35th International Conference on Computer Aided Verification*. 209–223.

[14] Sascha Böhme and Tjark Weber. 2010. Fast LCF-style proof reconstruction for Z3. In *Proceedings of the 1st International Conference on Interactive Theorem Proving*. 179–194.

[15] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. 2009. veriT: An open, trustable and efficient SMT-solver. In *Proceedings of the 22nd International Conference on Automated Deduction*. 151–156.

[16] Martin Bromberger, Jochen Hoenicke, and François Bobot. 2023. SMT-COMP 2023: Competition Report. Retrieved August 16, 2024 from https://smt-workshop.cs.uiowa.edu/2023/slides/smtcomp.pdf

[17] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT solver. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 150–153.

[18] Guillaume Bury. 2021. Dolmen: A validator for SMT-LIB and much more. In *Proceedings of the 19th International Workshop on Satisfiability Modulo Theories*. 32–39.

[19] Stefano Calzavara, Ilya Grishchenko, and Matteo Maffei. 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*. 47–62.

[20] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An interpolating SMT solver. In *Proceedings of the 19th International SPIN Workshop*. 248–254.

[21] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. 2017. Efficient certified RAT verification. In *Proceedings of the 26th International Conference on Automated Deduction*. 220–236.

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Proofs and refutations, and Z3. In *Proceedings of the 7th International Workshop on the Implementation of Logics*. 123–132.

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

[24] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. 2019. Ultimate TreeAutomizer (CHC-COMP tool description). In *Proceedings of the 6th Workshop on Horn Clauses for Verification and Synthesis*. 42–47.

[25] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into coq. In *Proceedings of the 29th International Conference on Computer Aided Verification*. 126–133.

[26] Gidon Ernst. 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*. 559–564.

[27] Gidon Ernst and José F. Morales. 2024. CHC-COMP 2024: Competition Report. Retrieved August 16, 2024 from https://chc-comp.github.io/2024/CHC-COMP2024Report-HCSV.pdf

[28] Pascal Fontaine, Jean-Yves Marion, Stephan Merz, Leonor Prensa Nieto, and Alwen Tiu. 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*. 167–181.

[29] Florian Frohn and Jürgen Giesl. 2022. Proving non-termination and lower runtime bounds with LoAT (system description). In *Proceedings of the 11th International Joint Conference on Automated Reasoning*. 712–722.

[30] Marco Gario and Andrea Micheli. 2015. PySMT: A solver-agnostic library for fast prototyping of SMT-based algorithms. In *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories*. 1–10.

[31] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. 2012. Synthesizing software verifiers from proof rules. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 405–416.

[32] Arie Gurfinkel and Nikolaj Bjørner. 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*. 6–10.

[33] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn verification framework. In *Proceedings of the 27th International Conference on Computer Aided Verification*. 343–361.

[34] Matthias Heizmann, Daniel Dietsch, Jochen Hoenicke, Alexander Nutz, Andreas Podelski, and Frank Schüssele. 2024. Ultimate Unihorn - Solver Description. DOI: https://doi.org/10.4204/EPTCS.402.10. Page 99.

[35] Marijn J.H. Heule, Warren A. Hunt, and Nathan Wetzler. 2013. Trimming while checking clausal proofs. In *Proceedings of the 13th Conference on Formal Methods in Computer-Aided Design*. 181–188.

[36] Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Nathan Wetzler. 2017. Efficient, verified checking of propositional proofs. In *Proceedings of the 8th International Conference on Interactive Theorem Proving*. 269–284.

[37] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. 2013. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction*. 345–359.

[38] C. A. R. Hoare. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12, 10 (1969), 576–580.

[39] Jochen Hoenicke and Tanja Schindler. 2022. A simple proof format for SMT. In *Proceedings of the 20th International Workshop on Satisfiability Modulo Theories*. 54–70.

[40] Hossein Hojjat, Philipp Rümmer, Pavle Subotic, and Wang Yi. 2014. Horn clauses for communicating timed systems. In *Proceedings of the 1st Workshop on Horn Clauses for Verification and Synthesis*. 39–52.

[41] Hossein Hojjat and Philipp Rümmer. 2018. The eldarica horn solver. In *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design*. 1–7.

[42] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A framework for verifying Java programs. In *Proceedings of the 28th International Conference on Computer Aided Verification*. 352–358.

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

[44] Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures - An Algorithmic Point of View* (2nd. ed.). Springer Berlin.

[45] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 179–191.

[46] Peter Lammich. 2020. Efficient verified (UN)SAT certificate checking. *Journal of Automated Reasoning* 64, 3 (2020), 513–532.

[47] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2021. RustHorn: CHC-Based verification for rust programs. *ACM Transactions on Programming Languages and Systems* 43, 4 (2021), 1–54.

[48] Rodrigo Otoni, Martin Blicha, Patrick Eugster, Antti E. J. Hyvärinen, and Natasha Sharygina. 2021. Theory-specific proof steps witnessing correctness of SMT executions. In *Proceedings of the 58th ACM/IEEE Design Automation Conference*. 541–546.

[49] Rodrigo Otoni, Martin Blicha, Patrick Eugster, and Natasha Sharygina. 2023. CHC model validation with proof guarantees. In *Proceedings of the 18th International Conference on integrated Formal Methods*. 62–81.

[50] Rodrigo Otoni, Matteo Marescotti, Leonardo Alt, Patrick Eugster, Antti Hyvärinen, and Natasha Sharygina. 2023. A solicitous approach to smart contract verification. *ACM Transactions on Privacy and Security* 26, 2 (2023), 1–28.

[51] Joseph E. Reeves, Benjamin Kiesl-Reiter, and Marijn J. H. Heule. 2023. Propositional proof skeletons. In *Proceedings of the 29th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 329–347.

[52] Andrew Reynolds and Hans-Jörg Schurr. 2024. AletheLF Checker (alfc). Retrieved August 16, 2024 from https://cvc5.github.io/docs/cvc5-1.1.2/proofs/output_alf.html

[53] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. 2015. On recursion-free horn clauses and Craig Interpolation. *Formal Methods In System Design* 47, 1 (2015), 1–25.

[54] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. 2017. A verified generational garbage collector for CakeML. In *Proceedings of the 8th International Conference on Interactive Theorem Proving*. 444–461.

[55] Hans-Jørg Schurr, Mathias Fleury, Haniel Barbosa, and Pascal Fontaine. 2021. Alethe: Towards a generic SMT proof format. In *Proceedings of the 7th Workshop on Proof eXchange for Theorem Proving*. 49–54.

[56] Carsten Sinz and Armin Biere. 2006. Extended resolution proofs for conjoining BDDs. In *Proceedings of the 1st International Symposium on Computer Science in Russia*. 600–611.

[57] Aaron Stump, Duckki Oe, Andrew Reynolds, Liana Hadarean, and Cesare Tinelli. 2013. SMT proof checking using a logical framework. *Formal Methods in System Design* 42, 1 (2013), 91–118.

[58] Ole Tange. 2011. GNU parallel - the command-line power tool. *;login: The USENIX Magazine* 36, 1 (2011), 42–47.

[59] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: A framework for abstraction refinement-based model checking. In *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*. 176–179.

[60] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. 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*. 422–429.

Received 16 August 2024; revised 15 January 2025; accepted 4 February 2025