Unsatisfiability Proofs for Horn Solving

Rodrigo Otoni[1]✉, Martin Blicha[1], Matias Barandiaran Rivera[2], Patrick Eugster[1], Jan Kofroň[3], and Natasha Sharygina[1]

[1] USI Lugano, Lugano, Switzerland
{otonir,blichm,eugstp,sharygin}@usi.ch
[2] Lancaster University, Leipzig, Germany
m.barandiaranrivera@lancaster.ac.uk
[3] Charles University, Faculty of Mathematics and Physics, Prague, Czech Republic
jan.kofron@d3s.mff.cuni.cz

In the proceedings of TACAS'25: https://doi.org/10.1007/978-3-031-90653-4_4

Abstract. Many verification tools currently rely on logic solvers as backend reasoning engines. Despite playing such a pivotal role, bugs are not uncommon in the complex codebases of these solvers. Validating their results is thus critical, with correctness witnesses often being used for this end. Output validation for constrained Horn clauses (CHC) solvers is not a well explored topic though, especially in regards to unsatisfiability results. This is a significant issue, given that CHC solvers are being increasingly employed in verification tooling. To address it, we propose an approach to validate CHC unsatisfiability results based on independently checkable proofs. Our approach is generic in regards to the solving algorithm, preprocessing steps, and exact proof format used, and works by first producing a coarse-grained proof during solving and then instantiating it into a suitable proof format by adding missing details, at which point the instantiated proof can be checked by an independent proof checker. We instrumented a state-of-the-art CHC solver to generate proofs in the Alethe format and performed a large-scale evaluation. Our results indicate that proofs can be produced with minimal overhead, can be efficiently checked, and have tractable sizes.

Keywords: Correctness witnesses · Horn clauses · Output validation.

1 Introduction

An interesting fragment of first-order logic (FOL) is that of constrained Horn clauses (CHC) [23]. Of particular note is the fact that the CHC fragment has been shown to be a match for Hoare logic [29], which makes it well suited to assist verification tasks [9]. To illustrate, CHC formulas have been used to aid in reasoning about the behaviours of procedural [9] and functional [21] programs, concurrent systems [31], and smart contracts [44].

Many logic solvers have been developed to enable automated reasoning of formulas in selected FOL fragments. The most common categories of such tools are arguably those of Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) solvers [37], which respectively solve formulas in the propositional

fragment of FOL and in extensions of it with theories such as arithmetics, arrays, and bit-vectors. Solvers for the CHC fragment also exist, e.g., Eldarica [32], FreqHorn [17], Golem [11], HoIce [13], MuCyc [51] and Z3-Spacer [36], being used, for instance, as the reasoning engines of verification tools targeting C/C++ [24], Java [33], Rust [39], and Solidity [1].

Despite their extensive usage in verification, logic solvers themselves are not immune to bugs. This issue has been well documented throughout the years by the reports of the annual SAT, SMT, and CHC competitions, which commonly encountered instances of competing state-of-the-art solvers disagreeing on certain benchmarks. In light of this, having guarantees about solvers' results is of critical importance, with such guarantees often coming in the form of correctness witnesses. The idea is that in addition to producing its standard output, a solver would also produce a witness that could be used by an independent checker to validate the given result. Many witness formats have been proposed for this end, aiming at validating outputs of SAT [3,15,25,27,28,47,49,52] and SMT [30,40,42,46,48,50] solvers. Concerns regarding bugs in the checkers can be addressed through codebase verification [26,38], which is better suited to checkers than to solvers since checkers' implementations are smaller and less complex.

The need for correctness witnesses in the context of CHC solving is clear, with the validation of results being a stated goal of the organisers of the CHC competition [16]. 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 suitable interpretations can be given to the predicates in such a way that all implications become logically valid. If this is possible, the input is considered satisfiable, SAT for short (not to be confused with Boolean satisfiability), and if it is not, the input is considered unsatisfiable, UNSAT for short. A possible witness for a SAT result could be obtained by computing a model, i.e., an interpretation of the predicates that makes all clauses logically valid, while a witness for an UNSAT result should demonstrate that this is not possible, entailing that *false* can be derived from the input.

While correctness witnesses have long been used to validate the results of SAT and SMT solvers, their use in validating CHC solvers' results has only recently started to be explored. The first approach to enable systematic validation of CHC results is that of the ATHENA framework [43], which uses CHC models to independently validate satisfiable results. When it comes to unsatisfiable results, however, no approach currently exists for independent validation to the best of our knowledge. To bridge this gap, we propose a novel validation approach based on the production, instantiation, and independent checking of unsatisfiability proofs. For a given input, our approach first produces a coarse-grained proof during solving and then instantiates it into a suitable proof format via the addition of details, e.g., hints, required by the format. The instantiated proof can then be checked by any independent checker catering to the proof format selected. One important note is that CHC solvers commonly employ intricate input preprocessing, which makes it necessary for the coarse proof produced to be backtranslated to the original input. The proposed solver workflow is shown

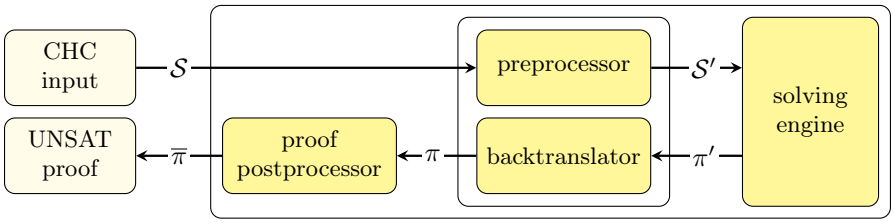


Fig. 1: Proposed workflow for a proof-producing Horn solver on an UNSAT input; $\mathcal{S}$ is the input formula and $\mathcal{S}'$ is a formula equisatisfiable to it, π and π' are proofs relating to $\mathcal{S}$ and $\mathcal{S}'$, and $\overline{\pi}$ is an instantiation of π to a specific proof format.

in Figure 1. It is generic in regards to the solving algorithm, preprocessing steps, and proof format used.

To evaluate our approach we instrumented the Golem solver to produce coarse proofs using the Spacer [36] solving algorithm, which is implemented in its namesake tool as well as in Golem and MuCyc, and to instantiate these proofs into the Alethe [46] proof format. We used all 600 integer benchmarks from the 2024 edition of the CHC competition in our experiments, with the Alethe proofs being independently checked by the Carcara [2] proof checker. The results we obtained indicate that (i) proof production and instantiation add minimal overhead to solving, (ii) proof checking can be done very efficiently, and (iii) proof sizes are tractable.

To summarise, our contributions are the following:

1) Proposal of an approach to validate unsatisfiability results of CHC solvers generically w.r.t. solving algorithm, preprocessing steps, and proof format.
2) Formalisation of a coarse-grained proof system and of how proofs in this system can be produced by the Spacer algorithm and be then backtranslated.
3) Description of how coarse proofs can be instantiated into the Alethe format.
4) Instrumentation of the Golem solver to produce coarse proofs and to then instantiate these proofs into the Alethe format.
5) Staging of a large scale evaluation to assess the approach's viability, using all 600 CHC-COMP 2024 integer benchmarks and the Carcara checker.

The remainder of the paper is structured as follows. Related work is discussed in Section 2. The necessary background is given in Section 3. Coarse-grained unsatisfiability proofs are described in Section 4. The production of coarse proofs with the Spacer algorithm is detailed in Section 5. The backtranslation of coarse proofs to account for preprocessing is discussed in Section 6. The instantiation of coarse proofs into the Alethe proof format is described in Section 7. Our evaluation is detailed in Section 8. Finally, our conclusions are given in Section 9.

2 Related Work

The production and checking of proofs to validate unsatisfiability results of logic solvers has been a topic of research for over a decade. Similar to how solver

development took place, proof formats were first proposed in the context of SAT solving and advances in the context of SMT solving then followed.

Even in the quite restricted domain of Boolean satisfiability, validation of unsatisfiability results is far from trivial. Many proof formats, offering different trade-offs between proof compactness and checking efficiency, have been proposed. Initial proposals were based on resolution [47] and clausal proofs [25], with resolution asymmetric tautology (RAT) [27] being the base of many follow-up developments, e.g., deletion RAT (DRAT) [52], linear RAT (LRAT) [15], and flexible RAT (FRAT) [3]. Recently, proofs based on propagation redundancy (PR) [28] and linear PR (LPR) [49] have also been proposed. Unsatisfiability proofs have been required in the SAT competition since its 2014 edition.

In the domain of SMT, proof formats are often attached to specific solvers. The ALETHE format [46] was designed for SMT proofs and was initially supported by VERIT [12], but has since also been integrated into CVC5 [4]. The CVC5 solver additionally caters for proofs based on the logical framework with side conditions (LFSC) [48] and the EUNOIA [50] logical framework. Individual unnamed proof formats are supported by SMTINTERPOL [30], OPENSMT [42], and Z3 [40]. As of the time of writing, unsatisfiability proofs are not a requirement in the main tracks of the SMT competition, with a proof exhibition track being used in recent editions to showcase existing formats.

Correctness witnesses are also being pursued in the broader context of formal verification, as exemplified by the addition of a validation track in the annual competition on software verification [7]. Many witness formats and validators have been proposed for software verification with varying degrees of success [8], with unsatisfiability proofs potentially leading to future developments.

3 Background

In this work we follow the standard constrained Horn clauses definition set forth by Rümmer et al. [45]. Given a first-order theory $\mathcal{T}$, i.e., a *background* theory, and a set of uninterpreted predicates $\mathcal{P} = \{p_1, p_2, \ldots\}$ disjoint from the signature of $\mathcal{T}$, a constrained Horn clause c is a formula

$$\forall V \cdot (B_1 \wedge \ldots \wedge B_k \wedge \varphi \implies H)$$

where

- V is the set containing all the variables appearing in the clause,
- B_i is an application $p(t_1, \ldots, t_n)$ of an n-ary predicate $p \in \mathcal{P}$ to terms of $\mathcal{T}$,
- φ is a (quantifier-free) formula from $\mathcal{T}$ over variables from V, and
- H is either an application $p(t_1, \ldots, t_n)$ of a predicate or the constant *false*.

It is common to refer to H as the clause's *head*, to $B_1 \wedge \ldots \wedge B_k \wedge \varphi$ as its *body*, and to φ as its *constraint*. A system of constrained Horn clauses $\mathcal{S} = \{c_1, \ldots, c_n\}$ is *satisfiable* if there exists a model $\mathcal{M}$ of $\mathcal{T}$ extended with interpretations for each predicate $p \in \mathcal{P}$ such that $\mathcal{M}$ satisfies c_i, written $\mathcal{M} \models c_i$, for every $c_i \in \mathcal{S}$.

Conversely, a CHC system is *unsatisfiable* if no such model exists, which can be proven by deriving *false* from $\mathcal{S}$ in some sound proof system.

For the remainder of the paper, without loss of generality, we assume that all predicates are only applied to variables; this greatly simplifies the presentation of CHC systems and unsatisfiability proofs, with normalisation steps being usually done before the preprocessing pipeline[4].

4 Coarse-Grained CHC Unsatisfiability Proofs

While there is no agreement on a single proof format for Horn solvers, there is a general understanding that a proof system based on *instantiation* and *resolution* rules can capture solver reasoning faithfully; this is illustrated by Gurfinkel (see Figure 6 in [22]) and by Björner et al. (see Section 2.2 in [9]). To achieve our goal of producing checkable proofs, we formalise a proof system based on a single rule that combines instantiation and hyper-resolution. We show later that this simple proof system is powerful enough to capture the common reasoning steps of Horn solvers. In the following we first describe our formalisation and then give an overview of the solver extensions required for proof production.

4.1 Single Derivation Rule

Given a CHC system $\mathcal{S}$, we consider a proof system with a single derivation rule

$$\sigma \models \varphi \ \frac{\sigma(B_1) \quad \ldots \quad \sigma(B_k) \quad \forall V \cdot (B_1 \wedge \ldots \wedge B_k \wedge \varphi \implies H)}{\sigma(H)} \,,$$

where $\forall V \cdot (B_1 \wedge \ldots \wedge B_k \wedge \varphi \implies H)$ is a clause from $\mathcal{S}$, σ is a total assignment of all the variables from V that satisfies φ, i.e., $\sigma \models \varphi$, and $\sigma(p(x_1, \ldots, x_n))$ denotes $p(\sigma(x_1), \ldots, \sigma(x_n))$, i.e., the result of replacing each variable $x \in V$ with its value assigned by σ.

Intuitively, the rule says that, given an instantiation of an input clause using σ that satisfies the constraint φ, if premises $\sigma(B_1), \ldots, \sigma(B_k)$ have been derived, then the conclusion $\sigma(H)$ can be derived. Note that the condition $\sigma \models \varphi$ is necessary for the rule to be sound, and that if there are no uninterpreted predicates in the input clause, then there are no additional premises and any assignment σ that satisfies φ can be used to derive $\sigma(H)$. A derivation of *false* from $\mathcal{S}$ using repeated applications of this derivation rule forms a proof of unsatisfiability of $\mathcal{S}$; we consider the *height* of a proof as the maximum number of rule applications on any path from the root to a leave.

[4]Normalisation of benchmarks is a standard procedure in the CHC competition, details are available at https://github.com/chc-comp/scripts/tree/master/format.

Example 1. Consider the following CHC system containing only one predicate (quantifiers are omitted for better readability):

$$\begin{aligned} x \leq 0 &\implies p(x) \\ p(x) \wedge x' = x + 1 &\implies p(x') \\ p(x) \wedge x \geq 1 &\implies \mathit{false} \end{aligned}$$

This CHC system is unsatisfiable as witnessed by the following proof of height 3.

$$\dfrac{\dfrac{\dfrac{}{x \leq 0 \implies p(x)}}{p(0)}_{x \mapsto 0} \qquad \dfrac{}{p(x) \wedge x' = x + 1 \implies p(x')}}{p(1)}_{\substack{x \mapsto 0 \\ x' \mapsto 1}} \qquad \dfrac{}{p(x) \wedge x \geq 1 \implies \mathit{false}}}{\mathit{false}}_{x \mapsto 1}$$

While these coarse proofs could, in theory, be automatically checked, no checker currently exists for this format. Instead of developing our own specialized proof checker, we show how coarse proofs can be instantiated in the Alethe format and be automatically checked by the proof checker Carcara.

4.2 Solver Extensions

A high-level architecture of state-of-the-art Horn solvers typically contains two main components: a *preprocessor* and a *backend solving engine*. The goal of the preprocessor is to make it easier for the backend engine to handle the input, and it works by transforming the input CHC system into a different CHC system that is simpler and *equisatisfiable*. A preprocessor typically implements several transformations that are statically or dynamically assembled into a *transformation pipeline*, based on the properties of the input clauses or the chosen backend engine. The goal of the backend engine is to decide the satisfiability of the transformed CHCs generated by the preprocessor. Importantly, if the preprocessor applies only equisatisfiable transformations, the result computed by the backend engine will also be valid for the original input CHC system.

To add proof-production capabilities to Horn solvers, three extensions to the architecture just described are required:

1. Augmentation of the backend engine to enable the production of coarse proofs using the single derivation rule described in Section 4.1.
2. Augmentation of the preprocessor with a *backtranslator* that is capable of *translating* coarse proofs for the transformed CHCs *back* into coarse proofs for the input CHCs.
3. Addition of a new *proof postprocessor* component to construct proofs in a chosen proof format, e.g., Alethe, from coarse proofs.

Some solvers, e.g., Eldarica and Z3-Spacer, already implement the first two extensions, at least partially, but no official publication describing the process exists at present. Despite this, it is important to note that *no* solver currently produces *independently* checkable proofs.

Given the variety of both preprocessing transformations and backend solving algorithms, we do not try to cover all of them here. Instead, we demonstrate how to augment a state-of-the-art solving algorithm and a widely used preprocessing technique to enable the production and backtranslation of coarse proofs, and we then show how such proofs can be instantiated into the Alethe format.

5 Production of Coarse Proofs in the Spacer Algorithm

The Spacer [35,36] algorithm is currently one of the most powerful algorithms for solving general systems of Horn clauses and, consequently, for program verification. Since its introduction, it has been extended with support for arrays [34], bit-vectors [19], and algebraic datatypes [20], as well as with global guidance to limit divergent behaviour [18]. The main implementation of Spacer is now a part of Z3 [41], but its success led, for instance, to independent reimplementations in the solvers Golem and MuCyc.

5.1 Brief Overview of Spacer

Spacer is often described as an abstract transition system with rules that update its state, with an alternative description based on induction [51] offering additional insight into its inner workings. With the goal of producing proofs in mind, we give here a more procedural description of the algorithm. This allows us to pinpoint the additional bookkeeping required to produce proofs with minimal overhead. We follow the original description of Spacer by Komuravelli et al. [36], but use the CHC terminology introduced in Section 3.

For its operation, the algorithm maintains two mappings, both from predicates and bounds to a set of formulas (the must- and may-summaries in [36]). A *must* mapping U keeps track, for each predicate p and a bound k, of the ground instances of p known to have a derivation of height at most k. The ground instances are tracked symbolically, rather than individually, as formulas that we call *must-derivations* (must-summaries in [36]). Similarly, a *may* mapping O keeps track of the ground instances that still may be derived, which we call *may-derivations* (may-summaries in [36]). Intuitively, given a predicate p and a bound k, if a ground instance of p is *not* in $\bigwedge O(p, k)$ then it *cannot* have a derivation of height at most k. Thus, U and O are, respectively, an underapproximation and an overapproximation of the ground instances that have bounded derivations.

The algorithm consists of two main parts: checking if a bounded derivation of *false* exists (`BndSafety` in [36]) and checking if a model has been discovered (`CheckInvariance` in [36]). The `CheckInvariance` part is responsible for pushing may-derivations to higher bounds and checking if a fixedpoint, i.e., a model,

Algorithm 1: `BndSafety`

Input : bound n
Global : formula mappings U and O, must-derivation graph G^M
Local : queue of proof obligations $\mathcal{Q}$

1 $\mathcal{Q}$.push($\langle false, n, true\rangle$)
2 **while** $\mathcal{Q}$ *is not empty* **do**
3 $\quad\langle p, k, \gamma\rangle \leftarrow \mathcal{Q}$.top()
4 $\quad C \leftarrow$ clauses with head p
5 $\quad$**foreach** $c \in C$ **do**
6 $\quad\quad$**if** *MustDerivable*(c, k, γ) **then**
7 $\quad\quad\quad$**if** $p = false$ **then return** UNSAFE
8 $\quad\quad\quad\mathcal{Q}$.pop()
9 $\quad\quad\quad$**continue**
10 $\quad$**foreach** $c \in C$ **do**
11 $\quad\quad$**if** *MayDerivable*(c, k, γ) **then**
12 $\quad\quad\quad\mathcal{Q}$.push(*Predecessor*$(c, k, \gamma)$)
13 $\quad$**if** *no new predecessor exists* **then**
14 $\quad\quad$*strengthen* O^p_k *to block* γ
15 $\quad\quad\mathcal{Q}$.pop()
16 **return** SAFE

Algorithm 2: `MustDerivable`

Input : $c \equiv \varphi \wedge B_1 \wedge \ldots \wedge B_n \implies H$, bound k, a constraint γ on ground instances of H

1 Let $p_1, \ldots, p_n, p$ be predicate symbols of $B_1, \ldots, B_n, H$
2 $\psi \leftarrow \varphi \wedge \bigvee U^{k-1}_{p_1} \wedge \ldots \wedge \bigvee U^{k-1}_{p_n}$
3 $(answer, M) \leftarrow SAT[\gamma \wedge \psi]$
4 **if** $answer = SAT$ **then**
5 $\quad\delta \leftarrow MBP(\gamma \wedge \psi, M)$
6 $\quad$Add δ to U^k_p
7 $\quad premises \leftarrow [\langle m_1, p_1\rangle, \ldots \langle m_n, p_n\rangle]$ such that $m_i \in U^{k-1}_{p_i}$ and $M \models m_i$
8 $\quad G^M$.add$(\langle \delta, p\rangle, c, premises)$
9 $\quad$**return** TRUE
10 **return** FALSE

Fig. 2: The `BndSafety` procedure and its `MustDerivable` subprocedure; the augmentation needed for proof production is highlighted.

has been found. We skip the explanation of this procedure, since it is not relevant to proof production, and instead focus on how `BndSafety` can leverage must-derivations to produce proofs.

5.2 Augmenting Spacer for Proof Production

The core idea is to track dependencies between learned must-derivations and to build a concrete proof at the end of the algorithm's execution based on the tracked information. The goal of `BndSafety`, shown in Figure 2, is to check if a derivation of *false* of height at most n exists, by using a priority queue of *proof obligations*. A proof obligation $\langle p, k, \gamma\rangle$ represents a question asking if there is a derivation of height at most k of some ground instance of p that satisfies γ. The initial proof obligation is $\langle false, n, true\rangle$ which represents the question asking if *false* has a derivation of size at most n. While some proof obligation $\langle p, k, \gamma\rangle$ remains, the algorithm attempts to resolve it. For that, it examines all clauses with head p and then checks if some derivation satisfying γ exists (using known must-derivations) or if it can prove such a derivation cannot exist (using known may-derivations). If neither is the case, then it generates new proof obligations (with decreased bound) from the failures of disproving the existence of the desired derivation with known may-derivations.

We skip the details regarding the refutation of proof obligations with may-derivations as well as the computation of predecessors and new may-derivations, as these are independent of proof production. Instead, we focus on the procedure `MustDerivable`, which takes a clause c, a bound k and a constraint γ on the

head H of c, i.e, a constraint over the variables of H, and determines if there is an instance of c such that

- the resulting ground instance of p satisfies γ, and
- the ground instances of the body predicates $p_1, \ldots, p_n$ are already known to be derivable.

This amounts to checking the satisfiability of γ with the body of c where uninterpreted predicates have been replaced by the disjunction of their known must-derivations. If the answer is SAT, a new must-derivation δ for p and bound k is computed using model-based projection (MBP) [10,36]. As an efficient under-approximation of quantifier elimination, MBP guarantees that every ground instance of p that satisfies δ can be derived in one step from some ground instances of $p_1, \ldots, p_n$ that satisfy $\bigvee U_{p_1}^{k-1}, \ldots, \bigvee U_{p_n}^{k-1}$, respectively. Thus, δ can be safely added to U_p^k.

In order to produce a proof of unsatisfiability, we only need to track which must-derivations of c's body predicates were needed for the new must-derivation of the head. To keep track of these dependencies, the algorithm should maintain a directed *hypergraph* of must-derivations. Each node consists of a must-derivation δ and a predicate symbol p. Each hyperedge $[\langle v_1, \ldots, v_n \rangle, v]$ is labeled with a clause c and represents the fact that the derivation of v can be obtained by hyper-resolution on the derivations of $v_1, \ldots, v_n$ and an appropriate instantiation of c. In the procedure `MustDerivable`, when a new must-derivation is learned, a new node with appropriate incoming edge is added to the graph. Note that each node will always have only a *single* incoming edge, but can participate in multiple outgoing edges. With the hypergraph of must-derivations to track dependencies, we can easily reconstruct a concrete proof after the algorithm finishes and determines that *false* can be derived.

5.3 Constructing Coarse Proofs from Must-Derivations

The hypergraph of must-derivations already defines the structure of the unsatisfiability proof, we only have to concretise individual nodes to ground instances. Algorithm 3 presents the construction of the proof.

The construction proceeds from the root (the derivation of *false*) towards the leaves, keeping a worklist of open goals. An open goal is a ground instance for which a derivation has not yet been constructed, together with a corresponding hypergraph node. The worklist is initialised with the ground instance *false* and the node $\langle \mathit{true}, \mathit{false} \rangle$. To avoid unnecessary multiple derivations of the same ground instance, the algorithm also keeps track of already encounted ground instances, i.e., instances that have already been derived or are already in the worklist. Then, while there is an open goal in the worklist, the algorithm closes it by constructing a derivation of the ground instance, possibly introducing new open goals. Each open goal $\langle \langle \delta, p \rangle, p(c_1, \ldots, c_n) \rangle$ from the worklist is processed as follows. From the graph, retrieve the single incoming edge labeled with clause $c \equiv \varphi \wedge B_1 \wedge \ldots \wedge B_n \implies H$ (with body predicates $p_1, \ldots, p_n$ and a head

Algorithm 3: Constructing Coarse Proofs from Must-Derivations

Input: must-derivation graph G^M
Local: worklist of open goals W, set of known ground instances S, proof π
$S \leftarrow \{\mathit{false}\}$, $W \leftarrow \{\langle\langle \mathit{true}, \mathit{false}\rangle, \mathit{false}\rangle\}$
while W *is not empty* **do**
  Pick and remove $\langle\langle \delta, p\rangle, p(c_1, \ldots, c_m)\rangle$ from W
  Let $e = [\langle\langle \delta_1, p_1\rangle \ldots, \langle \delta_n, p_n\rangle\rangle, \langle \delta, p\rangle]$ be the incoming edge for $\langle \delta, p\rangle \in G^M$
  Let $c = \varphi \wedge B_1 \wedge \ldots \wedge B_n \implies H$ be the label of e
  Compute σ, a satisfying assignment for $\varphi \wedge \delta_1 \wedge \ldots \wedge \delta_n$ such that $\sigma(H) = p(c_1, \ldots, c_m)$
  Add derivation $\sigma \dfrac{\sigma(B_1) \quad \ldots \quad \sigma(B_n) \quad \varphi \wedge B_1 \wedge \ldots \wedge B_n \implies H}{\sigma(H)}$ to π
  for $i \leftarrow 1 \ldots n$ **do**
    if $\sigma(B_i) \notin S$ **then**
      Add $\langle\langle \delta_i, p_i\rangle, \sigma(B_i)\rangle$ to W
      $S \leftarrow S \cup \{\sigma(B_i)\}$
return π

predicate symbol p) and the edge's source nodes $\langle \delta_1, p_1\rangle, \ldots, \langle \delta_n, p_n\rangle$. Compute a satisfying assignment σ for $\varphi \wedge \delta_1 \wedge \ldots \wedge \delta_n$ such that $\sigma(H) = p(c_1, \ldots, c_n)$. Note that this may, in general, require a call to an SMT solver, but such a satisfying assignment has to exist because of how δ and its premises were computed in `MustDerivable`. σ determines the ground instances $\sigma(B_1), \ldots, \sigma(B_n)$ for predicates $p_1, \ldots, p_n$ that form the premises of the derivation of $\sigma(H)$. Add the computed derivation of $\sigma(H)$ to the proof. Finally, add new entries $\langle \delta_i, p_i\rangle$ together with the corresponding ground instance $\sigma(B_i)$ to the worklist if this is the first time this ground instance has been encountered. When the worklist becomes empty no open goals are left and a derivation of *false* from the input clauses, i.e., a coarse unsatisfiability proof, has been constructed.

6 Backtranslation of Coarse Proofs

Preprocessing is a vital part of state-of-the-art Horn solvers. It rewrites the input CHC system into a new system with the goal to simplify the input and make it more suitable for the backend solving engine. In practice, a good preprocessing can significantly improve the performance of the solving algorithm. Typically, a preprocessor implements several transformations that we refer to as *passes*. Multiple passes are applied in sequence, each receiving its input system from the previous stage and forwarding the transformed system to the next stage.

6.1 Coupling Transformation Passes with Backtranslators

In order to support proof production, each transformation pass in the preprocessor has to be coupled with a *backtranslator*, which is able to *translate* a proof for the transformed system *back* to a proof for the input system.

Definition 1. *Given a transformation pass τ that takes an input CHC system $\mathcal{S}$ and produces an equisatisfiable system $\mathcal{S}'$, a* backtranslator *computes a proof of unsatisfiability π for $\mathcal{S}$ from a proof of unsatisfiability π' for $\mathcal{S}'$.*

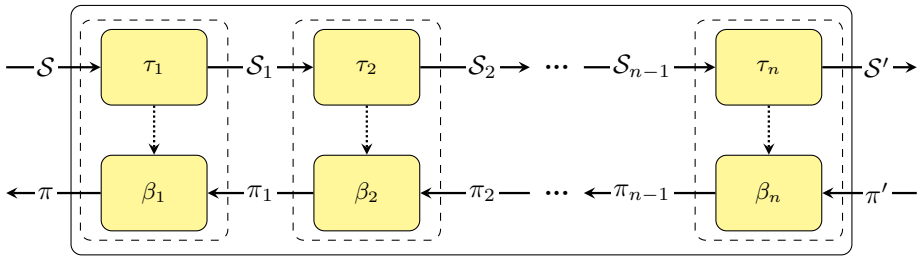


Fig. 3: Workflow of the preprocessor and backtranslator on an unsatisfiable input; every preprocessing step τ_i must be matched with a backtranslation step β_i.

When a proof-producing Horn solver couples each preprocessing pass with the corresponding backtranslator, a proof computed from the backend engine can be fed backwards through the sequence of backtranslators to obtain a proof for the original system. This process is depicted in Figure 3. Each preprocessing step τ_i provides the necessary information regarding the applied transformation to the coupled backtranslator β_i. This enables the backtranslator to apply the necessary modifications to the proof.

There are many different transformations a solver can employ. We demonstrate the concept of a backtranslator for a common transformation technique that eliminates an uninterpreted predicate from the CHC system, and then briefly discuss some other transformations.

6.2 Predicate Elimination Transformation

Eliminating predicates from the input CHC system (the *unfold* operation in [9]) is a common technique that can greatly improve the performance of the backend algorithm. Suppose that p is an uninterpreted predicate occurring in the system $\mathcal{S}$ such that it is not present in both body and head of the same clause. Moreover, for simplicity, assume that p does not occur multiple times in the body of any clause. Then, $\mathcal{S}$ can be transformed to an equisatisfiable system $\mathcal{S}'$ without p by

1. Collecting separately the clauses where p occurs in the body and the head.
   - $C_{in} = \{c \in \mathcal{S} \mid p \text{ is the head of } c\}$
   - $C_{out} = \{c \in \mathcal{S} \mid p \text{ is in the body of } c\}$
2. Resolving every pair of clauses from the Cartesian product $C_{in} \times C_{out}$.
3. Removing all the clauses in $C_{in} \cup C_{out}$ from $\mathcal{S}$ and adding all their resolvents to obtain $\mathcal{S}'$.

Note that the second step may require variable renaming to ensure there are no clashes and that resolving on the occurrence of p is sound.

6.3 Backtranslator for Predicate Elimination

The backtranslator corresponding to a predicate elimination pass must remember, for each resolvent, which pair of clauses was used to derive the resolvent. Then, when the backtranslator receives a proof, it checks for existence of steps using an instantiation of some resolvent. Each such step is then replaced with two steps using instantiations of the two original clauses. For ease of presentation, we consider the case of linear clauses, i.e., clauses with a single predicate in their body. Suppose clauses $Q \wedge \varphi_1 \implies P$ and $P \wedge \varphi_2 \implies R$ were replaced by a clause $Q \wedge \varphi_1 \wedge \varphi_2 \implies R$, and that this clause has been instantiated in the proof π' using assignment σ, leading to a derivation of the ground instance $\sigma(R)$. Then, σ defines the ground instance of P that can be derived from the first clause and resolving the obtained ground instance of P with the instantiation of the second clause (again using σ) yields the desired ground instance $\sigma(R)$.

Example 2. Assume a predicate p is present in the following clauses:

- $C_{in} = \left\{ \begin{array}{r} q(x,b) \wedge s(y) \wedge b = true \wedge z = x + y \implies p(z,y), \\ q(x,b) \wedge b = false \implies p(x,x) \end{array} \right\}$
- $C_{out} = \left\{ p(x,y) \wedge x' = x + 1 \wedge y' = y \implies r(y',x') \right\}$

Then we get two resolvents:

- $q(x,b) \wedge s(y) \wedge b = true \wedge z = x + y \wedge x' = z + 1 \wedge y' = y \implies r(y',x')$
- $q(x,b) \wedge b = false \wedge x' = x + 1 \wedge y' = x \implies r(y',x')$

Suppose that the proof π' for $\mathcal{S}'$ contains a step with the second resolvent.

$$\dfrac{\begin{array}{l} x \mapsto 0,\ b \mapsto false \\ x' \mapsto 1,\ y' \mapsto 0 \end{array} \quad q(0,false) \qquad q(x,b) \wedge b = false \wedge x' = x + 1 \wedge y' = x \implies r(y',x')}{r(0,1)}$$

The backtranslator then splits this step into two steps that will use the two original clauses and an intermediate derived fact $p(0,0)$:

$$\dfrac{\begin{array}{l} x \mapsto 0 \\ b \mapsto false \end{array} \quad q(0,false) \qquad q(x,b) \wedge b = false \implies p(x,x) \qquad \begin{array}{r} p(x,y) \wedge x' = x + 1 \wedge \\ y' = y \implies r(y',x') \end{array}}{\begin{array}{l} x \mapsto 0,\ y \mapsto 0 \\ x' \mapsto 1,\ y' \mapsto 0 \end{array} \quad \dfrac{p(0,0)}{r(0,1)}}$$

6.4 Other Transformation Passes

There are many possible transformation passes beyond predicate elimination, often with trivial or almost trivial backtranslators. For example, removing *redundant* clauses from the system has a trivial backtranslator that does nothing, as the proof for the transformed system is a valid proof for the original system. Similarly, *rewriting a constraint* of some clause to an equisatisfiable constraint does not require a backtranslator to make any structural changes to a proof. The backtranslator only needs to know how to adjust the satisfying assignment σ used for instantiation to obtain a satisfying assignment of the original constaint while yielding the same ground instances of the predicates.

7 From Coarse Proofs to Alethe Proofs

The Alethe proof format [46] has been developed in the context of SMT, originally to capture the reasoning steps of the SMT solver veriT [12], but aspiring to become a proof format for SMT solvers in general. The specification of Alethe [5] is an evolving document, but it has been relatively stable for some time and there exists an independent checker, Carcara [2], for Alethe proofs.

Compared to our coarse proof format, Alethe has been designed to capture a large variety of ways of reasoning a SMT solver can implement, covering everything from rewriting, through Boolean reasoning, to reasoning in various SMT theories and reasoning about quantifiers. It currently consists of 91 rules that capture very fine-grained reasoning steps. The advantage of having fine-grained rules is that a concrete rule application can be easily checked for correctness, by checking if all premises have been derived and if the claimed conclusion matches the expected one.

We show how an Alethe proof can be constructed from a coarse CHC unsatisfiability proof. For this, we assume the CHC problem is given in the input format of CHC-COMP [14], which is a specific fragment of the SMT-LIB language version 2.6 [6]. Assume there is an input CHC system $\mathcal{S} = \{c_1, \ldots, c_n\}$ and a coarse proof π deriving *false* from $\mathcal{S}$. An Alethe proof is a sequence of *steps* where an initial prefix of the sequence consists of the formulas from the corresponding satisfiability problem. These are called *assumption* steps in Alethe. Formulas in the assumption steps can simply be copied from the assertions in the input file. Afterwards, each derivation step from the coarse proof is expanded into a sequence of Alethe steps.

Consider a single derivation step in the coarse proof. To expand it into a sequence of Alethe steps, we first split the derivation step into instantiation and resolution steps. Alethe has a `resolution` rule that matches the resolution in a coarse proof. Instantiation, on the other hand, is more complicated. In the coarse proof, we not only instantiate input clauses, but also discharge the clauses' constraint at the same time. In Alethe, this has to be done separately. The instantiation rule of Alethe, denoted as `forall_inst`, only performs syntactic substitution, it does not perform any simplifications. The simplification steps have to be added explicitly. Alethe has several rules to capture arithmetic and boolean simplifications, e.g., `sum_simplify`, `minus_simplify`, `prod_simplify`, `and_simplify`, `or_simplify`. How exactly to connect all the simplification steps is best demonstrated on an example.

Example 3. Consider the derivation step from Example 1, that derives $p(1)$ from $p(0)$ and the input clause $p(x) \wedge x' = x + 1 \implies p(1)$, using the assignment $\sigma\colon x \mapsto 0, x' \mapsto 1$. A possible expansion of this step in Alethe is shown in Figure 4. Steps 1 and 11 correspond to the premises of the coarse proof step and step 25 corresponds to its conclusion. Steps 12 to 24 are the fine-grained expansion of the coarse step using Alethe rules. They capture the fact that the constraint $x' = x + 1$ after instantiation to $1 = 0 + 1$ simplifies to *true*. In addition to the simplifications, some steps are necessary to deal with the

1. $\forall x, x' \cdot p(x) \wedge x' = x + 1 \implies p(x')$ (input clause)

⋮

11. $p(0)$ (...)
12. $\neg(\forall x, x' \cdot p(x) \wedge x' = x + 1 \implies p(x')) \vee$
    $(p(0) \wedge (1 = (0 + 1)) \implies p(1))$ (forall_inst, $x \mapsto 0$, $x' \mapsto 1$)
13. $\neg(\forall x, x' \cdot p(x) \wedge x' = x + 1 \implies p(x')), p(0) \wedge 1 = (0 + 1) \implies p(1)$ (or, 12)
14. $p(0) \wedge (1 = (0 + 1)) \implies p(1)$ (resolution, 1,13)
15. $\neg(p(0) \wedge (1 = (0 + 1))), p(1)$ (implies, 14)
16. $1 = (0 + 1)$ (sum_simplify)
17. $(1 = (0 + 1)) \iff (1 = 1)$ (cong, 16)
18. $(1 = 1) \iff \mathit{true}$ (eq_simplify)
19. $(1 = (0 + 1)) \iff \mathit{true}$ (trans, 17,18)
20. $p(0) \wedge (1 = (0 + 1)) \iff p(0) \wedge \mathit{true}$ (cong, 19)
21. $p(0) \wedge \mathit{true} \iff p(0)$ (and_simplify)
22. $p(0) \wedge (1 = (0 + 1)) \iff p(0)$ (trans, 20,21)
23. $p(0) \wedge (1 = (0 + 1)), \neg p(0)$ (equiv2, 22)
24. $p(0) \wedge (1 = (0 + 1))$ (resolution, 11,12)
25. $p(1)$ (resolution, 15,24)

Fig. 4: Example of an ALETHE proof.

fundamental design of ALETHE, which sees every derived formula as a "clause", with comma separating the "literals". Note the most of our steps are unit clauses in this sense, with the exception of steps 13, 15 and 23. Each step is annotated with the applied ALETHE rule and a (possibly empty) list of premises used.

In general, a derivation of form $\sigma \dfrac{\sigma(B_1) \quad \ldots \quad \sigma(B_n) \quad \varphi \wedge B_1 \wedge \ldots \wedge B_n \implies H}{\sigma(H)}$ is expanded by following five stages. First, instantiate the clause to $\sigma(\varphi) \wedge \sigma(B_1) \wedge \ldots \wedge \sigma(B_n) \implies \sigma(H)$. Second, prove that $\sigma(\varphi) \iff \mathit{true}$. Third, use this fact to show that $\sigma(B_1) \wedge \ldots \wedge \sigma(B_n) \iff \sigma(\varphi) \wedge \sigma(B_1) \wedge \ldots \wedge \sigma(B_n)$. Fourth, resolve with the first n premises to derive the instantiated body $\sigma(\varphi) \wedge \sigma(B_1) \wedge \ldots \wedge \sigma(B_n)$. Finally, resolve this with the instantiated last premise to obtain the conclusion $\sigma(H)$. Expanding every step of the coarse proof in this manner results in a full ALETHE proof without holes that derives *false* from the input CHCs.

Note that one step of the coarse proof expands to a large number of ALETHE steps. Just proving that $\sigma(\varphi) \iff \mathit{true}$ requires number of steps that is linear with respect to the number of subterms of φ, as, intuitively, the proof computes the constant value of each subterm. To avoid blowup in the size of the textual representation, terms can be named using the SMT-LIB attribute `:named` and the assigned names can be used instead of full string representation of the terms.

8 Evaluation

To assess the suitability of our approach to enable the independent validation of UNSAT results of Horn solvers, we aim to answer three research questions:

- **RQ1** Can unsatisfiability proofs serve as correctness witnesses?
- **RQ2** Is the overhead of producing and checking proofs manageable?
- **RQ3** Are proof sizes a limiting factor?

In the following we describe the benchmarks and tools used in our experiments, in Section 8.1, and the results we obtained, in Section 8.2.

8.1 Benchmarks and Tools

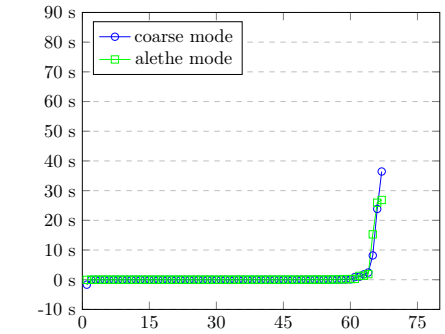
We used all benchmarks of the two linear integer arithmetic (LIA) tracks of CHC-COMP 2024 [16], referred to here as LIA-lin, consisting of benchmarks containing only linear Horn clauses, and LIA-nonlin, consisting of benchmarks containing nonlinear Horn clauses; a clause is linear if it has at most one predicate in its body and is nonlinear otherwise. Each track had 300 benchmarks, making for a total of 600 benchmarks used in our evaluation.

For proof production, we instrumented the Golem solver with the changes described in the previous sections to be able to output both coarse and Alethe proofs for UNSAT results. We chose Golem because it is both a very efficient solver, performing very well in recent CHC-COMP editions, and has a codebase that can be easily extended. For checking the Alethe proofs produced, we used the latest stable code of the Carcara checker (commit 9053d18).

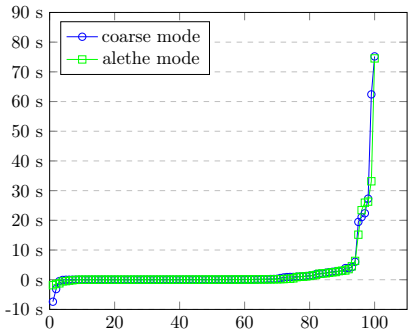
8.2 Results

To answer our research questions, we executed the instrumented solver in three different modes for each benchmark: default mode, without proof production, coarse mode, with production of coarse proofs, and alethe mode, with production of coarse proofs and instantiation of them into the Alethe format. None of Golem's optimisations or preprocessing passes had to be disabled for the proof-producing modes. A Linux machine with 64 AMD EPYC 7452 processors and 256 GB of memory was used for the evaluation. All individual tool executions had a timeout of 30 minutes and a memory limit of 10 gigabytes.

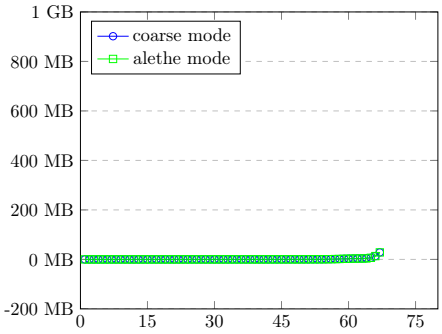
RQ1 The executions in default mode yielded 67 LIA-lin and 100 LIA-nonlin UNSAT results. The respective 167 benchmarks also all yielded UNSAT results in executions in alethe mode and produced Alethe proofs; no benchmark with an UNSAT result in default mode had a timeout in alethe mode. The proofs were forwarded to Carcara and were all successfully validated. This shows that the production and independent checking of unsatisfiability proofs can be made practical with off-the-shelf tools.



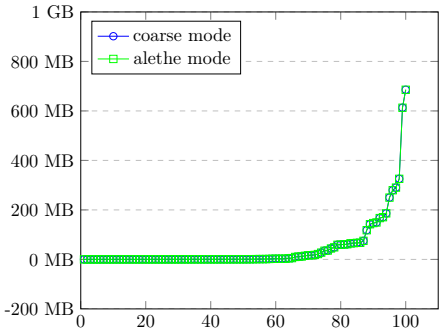
(a) LIA-lin runtime overheads.



(b) LIA-nonlin runtime overheads.

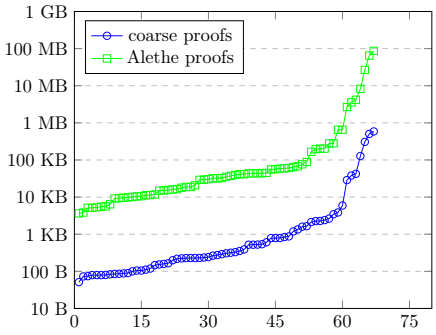


(c) LIA-lin memory overheads.

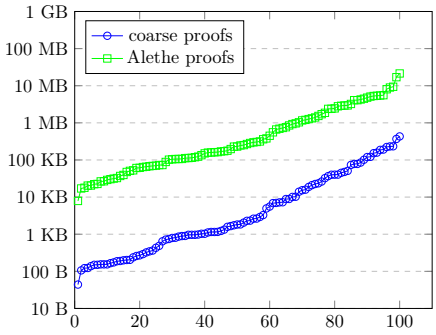


(d) LIA-nonlin memory overheads.

Fig. 5: Overheads of the coarse and alethe modes in relation to the default mode. The overheads are ordered according to their size, with the x-axis indicating their position in the order and the y-axis indicating their size.



(a) LIA-lin proofs.



(b) LIA-nonlin proofs.

Fig. 6: Sizes of the proofs produced in the coarse and alethe modes. The proofs are ordered according to their size, with the x-axis indicating their position in the order and the y-axis indicating their size (in logarithmic scale).

RQ2 We gathered the overhead, both in runtime and memory usage, of the coarse and alethe modes in relation to the default mode. The results can be seen in Figure 5. For the majority of benchmarks the overhead in both metrics was negligible, with the few outliers registered not surpassing a slowdown of 90 seconds and an increase of 800 MB of memory usage. The small negative overheads observed in runtime are likely due to process scheduling during the experiments. Also of note is the fact that, in addition to having a small overhead in relation to the default mode, the coarse and alethe modes had a small difference between themselves, indicating that the instantiation of coarse proofs into the Alethe format does not incur a significant computation cost. In regards to proof checking, it was very efficient, with the time taken to check all 167 proofs being 58 seconds and the average memory footprint of Carcara being 17 MB. This shows that the production and checking of unsatisfiability proofs does not significantly impact solver performance.

RQ3 We gathered the sizes of all proofs produced. They can be seen in Figure 6. As is to be expected, Alethe proofs are larger than coarse proofs, in most cases by an order of magnitude. Despite this difference, the absolute size of each proof is relatively small, never going over 100 MB. This shows that proof sizes are not likely to be a limiting factor in output validation.

9 Conclusions

We presented a validation approach for UNSAT results of CHC solvers based on unsatisfiability proofs. Our approach is generic w.r.t. to the solving algorithm, preprocessing steps, and proof format used, and was implemented in the Golem solver. A large scale evaluation was conducted and its results indicate that validation of UNSAT results can be made practical with limited overhead being added to solving performance.

Directions for future work include the instantiation of coarse proofs to different first-order theories and proof formats, and the investigation of how well our coarse proofs lend themselves to acceleration techniques for CHC solving.

Acknowledgments. This work was supported by the Swiss National Science Foundation via grants 200021_185031 and 2000021_197353, by the Czech Science Foundation via grant 23-06506S, and by the Hasler Foundation via its Responsible AI Programme.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Alt, L., Blicha, M., Hyvärinen, A., 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. 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)
4. 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)
5. Barbosa, H., Fleury, M., Fontaine, P., Schurr, H.J.: The Alethe Proof Format: An Evolving Specification and Reference, https://verit.loria.fr/documentation/alethe-spec.pdf, Accessed: 2024-10-07
6. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017)
7. Beyer, D.: State of the Art in Software Verification and Witness Validation: SV-COMP 2024. In: Proceedings of the 30th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 299–329 (2024)
8. 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)
9. 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)
10. Bjørner, N., Janota, M.: Playing with Quantified Satisfaction. In: Proceedings of the 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. pp. 15–27 (2015)
11. Blicha, M., Britikov, K., Sharygina, N.: The Golem Horn Solver. In: Proceedings of the 35th International Conference on Computer Aided Verification. pp. 209–223 (2023)
12. 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)
13. Champion, A., Kobayashi, N., Sato, R.: HoIce: An ICE-Based Non-linear Horn Clause Solver. In: Proceedings of the 16th Asian Symposium on Programming Languages and Systems. pp. 146–156 (2018)
14. CHC-COMP Benchmark Format, https://chc-comp.github.io/format.html, Accessed: 2024-10-07
15. 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)
16. Ernst, G., Morales, J.F.: CHC-COMP 2024 Report. https://chc-comp.github.io/CHC-COMP2024Report-HCSV.pdf, Accessed: 2024-10-07
17. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling Invariants from Frequency Distributions. In: Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design. pp. 100–107 (2017)

18. Govind V. K., H., Chen, Y., Shoham, S., Gurfinkel, A.: Global Guidance for Local Generalization in Model Checking. In: Proceedings of the 32nd International Conference on Computer Aided Verification. pp. 101–125 (2020)
19. Govind V. K., H., Fedyukovich, G., Gurfinkel, A.: Word Level Property Directed Reachability. In: Proceedings of the 2020 IEEE/ACM International Conference on Computer-Aided Design. pp. 1–9 (2020)
20. Govind V. K., H., Shoham, S., Gurfinkel, A.: Solving Constrained Horn Clauses modulo Algebraic Data Types and Recursive Functions. Proceedings of the ACM on Programming Languages **6**(POPL) (2022)
21. 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)
22. Gurfinkel, A.: Program Verification with Constrained Horn Clauses (Invited Paper). In: Proceedings of the 34th International Conference on Computer Aided Verification. pp. 19–29 (2022)
23. 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)
24. 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)
25. 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)
26. Heule, M.J.H., 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)
27. 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)
28. Heule, M.J.H., Kiesl, B., Biere, A.: Strong Extension-Free Proof Systems. Journal of Automated Reasoning **64**(3), 533–554 (2020)
29. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM **12**(10), 576–580 (1969)
30. 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)
31. 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)
32. 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)
33. 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)
34. Komuravelli, A., Bjørner, N., Gurfinkel, A., McMillan, K.L.: Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. In: Proceedings of the 15th Conference on Formal Methods in Computer-Aided Design. pp. 89–96 (2015)
35. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. In: Proceedings of the 26th International Conference on Computer Aided Verification. pp. 17–34 (2014)

36. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-Based Model Checking for Recursive Programs. Formal Methods in System Design **48**(3), 175–205 (2016)
37. Kroening, D., Strichman, O.: Decision Procedures - An Algorithmic Point of View. Springer Berlin, Heidelberg (2016)
38. Lammich, P.: Efficient Verified (UN)SAT Certificate Checking. Journal of Automated Reasoning **64**(3), 513–532 (2020)
39. 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)
40. 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)
41. 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)
42. Otoni, R., Blicha, M., Eugster, P., Hyvärinen, A., 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)
43. Otoni, R., Blicha, M., Eugster, P., Sharygina, N.: CHC Model Validation with Proof Guarantees. In: Proceedings of the 18th International Conference on integrated Formal Methods. pp. 62–81 (2023)
44. 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)
45. Rümmer, P., Hojjat, H., Kuncak, V.: On Recursion-Free Horn Clauses and Craig Interpolation. Formal Methods In System Design **47**(1), 1–25 (2015)
46. 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)
47. 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)
48. 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)
49. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: Verified Propagation Redundancy and Compositional UNSAT Checking in CakeML. International Journal on Software Tools for Technology Transfer **25**(2), 167–184 (2023)
50. The authors of cvc5: Cooperating Proof Calculus. https://cvc5.github.io/docs/cvc5-1.2.0/proofs/output_cpc.html, Accessed: 2024-10-07
51. Tsukada, T., Unno, H.: Inductive Approach to Spacer. Proceedings of the ACM on Programming Languages **8**(PLDI) (2024)
52. 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)