Structure-Guided Solution of Constrained Horn Clauses

Omer Rappoport(✉), Orna Grumberg, and Yakir Vizel

Technion - Israel Institute of Technology, Haifa, Israel
{omer.r,orna,yvizel}@cs.technion.ac.il

Abstract. We present StHorn, a novel technique for solving the satisfiability problem of CHCs, which works lazily and incrementally and is guided by the structure of the set of CHCs. Our technique is driven by the idea that a set of CHCs can be solved in parts, making it an easier problem for the CHC-solver. Furthermore, solving a set of CHCs can benefit from an interpretation revealed by the solver for its subsets. Our technique is *lazy* in that it gradually extends the set of checked CHCs, as needed. It is *incremental* in the way it constructs a solution by using satisfying interpretations obtained for previously checked subsets. In order to capture the structure of the problem, we define an *induced CHC hypergraph* that precisely corresponds to the set of CHCs. The paths in this graph are explored and used to select the clauses to be solved.

We implemented StHorn on top of two CHC-solvers, Spacer and Eldarica. Our evaluation shows that StHorn complements both tools and can solve instances that cannot be solved by the other tools. We conclude that StHorn can improve upon the state-of-the-art in CHC solving.

Keywords: Constrained Horn Clauses · CHC-SAT · Verification

1 Introduction

Constrained Horn Clauses (CHCs) is a fragment of First Order Logic (FOL) that has gained much attention in recent years. One main reason for the rising interest in CHCs is the ability to reduce many verification problems to satisfiability of CHCs [5,7,11,17,18,20,25,33]. For example, program verification can naturally be described as the satisfiability of CHCs modulo a background theory such as Linear Integer Arithmetic [7]. CHC-solvers can be used as the back-end for a variety of verification tools [15,19,27,30], separating the generation of verification conditions from the decision procedure that determines their correctness.

In this paper we present StHorn, a novel, structure-guided, lazy and incremental technique for solving the satisfiability problem of CHCs modulo a background theory. Our technique is driven by the idea that a set of CHCs can be solved in parts, making each sub-problem easier to solve. Furthermore, solving a set

This research is partially funded by the Israel Science Foundation (ISF), grant no. 2875/21.

É. André and J. Sun (Eds.): ATVA 2023, LNCS 14216, pp. 117–138, 2023.
https://doi.org/10.1007/978-3-031-45332-8_6

of CHCs can benefit from satisfying interpretations, which are revealed when handling its subsets.

StHorn uses an existing CHC-solver [12,14,23,24,28,34] as a "black-box". Given a set of CHCs Π, it chooses a subset of CHCs and tries to solve it using the existing CHC-solver. If it finds that the subset is unsatisfiable, then it concludes the entire set of CHCs is unsatisfiable. Otherwise, if a satisfying interpretation is found, StHorn extends the subset of CHCs, adapts the satisfying interpreation to be consistent with the new extended subset, and reinvokes the CHC-solver on the extended subset of CHCs. This process continues iteratively until either a subset is found to be unsatisfiable, or a satisfying interpretation is found for the entire set of CHCs Π.

There are three pillars to StHorn: (i) the *structure-guided* selection of CHC subsets to be processed; (ii) the *incremental* usage of satisfying interpretations when solving the different subsets of CHCs; and lastly (iii) the *lazy* processing of CHCs only when needed (when none of the processed subsets is unsatisfiable).

In order to capture the structure of the problem, we define an *induced CHC hypergraph* that precisely corresponds to the set of CHCs and depicts the dependencies between them. We present an algorithm for finding the shortest nontrivial hyperpath in the graph. This algorithm is used for selecting the CHC subsets to be solved, resulting in minimal clause addition at each iteration. Our selection strategy is based on the understanding that solving small subsets is often easier and can be advantageous to the overall solution.

To be incremental, StHorn maintains an interpretation that is injected into the CHC-solver as a starting point at each iteration, enabling the solver to search in a reduced state space. When a subset of CHCs is extended, it must be ensured that the existing satisfying interpretation is consistent with the extended subset. To this end, StHorn implements an amending procedure, which receives a set of CHCs and an interpretation, which might not satisfy all of them, and amends the interpretation such that it becomes consistent with the extended subset.

The combination of the choice of the examined subsets and the way in which the interpretation is amended defines how StHorn guides the search for a satisfying interpretation. Intuitively, StHorn guides the search based on the structure of the CHCs as reflected by the induced CHC hypergraph.

We implemented StHorn on top of two CHC-solvers: Spacer [28] and Eldarica [24]. For evaluation, we used the CHC-COMP'22 [13] benchmarks, and compared StHorn against Spacer and Eldarica. Our evaluation shows that StHorn complements both tools and can solve instances that cannot be solved by the other tools. We conclude that StHorn can improve upon the state-of-the-art in CHC solving.

The main contributions of this work are as follows:

- We develop an efficient technique for solving CHCs, which considers the structure of the CHCs during the search for a solution.
- The search for a solution is done incrementally, based on interpretations learned in previous iterations.

– We implemented a generic framework that can be used with any existing CHC-solver. In addition, we implemented two instances of StHorn: one using Spacer and the other that uses Eldarica and evaluated their performance. Our implementation is open-source and publicly available.

2 Preliminaries

We consider first-order logic (FOL) modulo a background theory $\mathcal{T}$ and denote it by FOL($\mathcal{T}$). We adopt the standard notation and terminology, where FOL($\mathcal{T}$) is defined over a signature Σ that consists of constant, predicate and function symbols, some of which may be interpreted by $\mathcal{T}$. The set of uninterpreted predicate symbols in Σ is denoted by $\mathcal{P}$. From now on, we fix the background theory $\mathcal{T}$.

A *p-formula* is an application of the form $p(t_1, \ldots, t_n)$ for some predicate symbol $p \in \mathcal{P}$ and first-order terms t_i. Given a set $\mathcal{S}$ of symbols, a formula φ is *$\mathcal{S}$-free* if no $\mathcal{S}$ symbols occur in φ. We write $\varphi[X]$ for a formula φ with free variables X. We use $\top$ and $\bot$ to represent the constant symbols True and False, respectively.

2.1 Constrained Horn Clauses

Definition 1. *A* Constrained Horn Clause *(*CHC *or* clause*) is a FOL formula π of the form $\forall X.(B[X] \rightarrow H[X])$, where*

– *$H[X]$, denoted* head(π)*, is either a p-formula for some $p \in \mathcal{P}$, or is $\mathcal{P}$-free.*
– *$B[X]$, denoted* body(π)*, is a formula either of the form $\psi_1 \wedge \cdots \wedge \psi_k \wedge \varphi$ or φ, where each ψ_i is a p-formula for some $p \in \mathcal{P}$, and φ is a $\mathcal{P}$-free constraint.*

A clause is called a *query* if its head is $\mathcal{P}$-free; otherwise, it is called a *rule*. A rule with $\mathcal{P}$-free body is called a *fact*. A clause is *linear* if its body contains at most one predicate symbol from $\mathcal{P}$; otherwise, it is *non-linear*. We refrain from explicitly adding the universal quantifier when the set of variables is clear from the context.

A set of CHCs Π is *satisfiable* if there exists an interpretation of the uninterpreted predicate symbols in $\mathcal{P}$ such that each CHC π in Π is valid under the interpretation (modulo $\mathcal{T}$). CHC-solvers attempt to determine the satisfiability of a set of CHCs by searching for a satisfying interpretation that is definable in $\mathcal{T}$. Such an interpretation is called a $\mathcal{T}$-interpretation. Formally, a *$\mathcal{T}$-interpretation* $\mathcal{I}$ associates every $p \in \mathcal{P}$ with a $\mathcal{P}$-free formula $\mathcal{I}(p)$ over the signature Σ of FOL($\mathcal{T}$). Given a CHC π and a $\mathcal{T}$-interpretation $\mathcal{I}$, we denote by $\mathcal{I}(\pi)$ the formula obtained after substituting every p-formula that occurs in π with $\mathcal{I}(p)$. A $\mathcal{T}$-interpretation $\mathcal{I}$ *satisfies* a CHC π, denoted $\mathcal{I} \models \pi$, if $\mathcal{I}(\pi)$ is valid (modulo $\mathcal{T}$). A $\mathcal{T}$-interpretation $\mathcal{I}$ satisfies a set of CHCs Π, denoted $\mathcal{I} \models \Pi$, if $\mathcal{I} \models \pi$ for every $\pi \in \Pi$. Note that, if there exists a satisfying $\mathcal{T}$-interpretation for Π, then Π is satisfiable. The converse, however, may not hold due to the limited expressiveness of FOL($\mathcal{T}$). Henceforth, we will only consider $\mathcal{T}$-interpretations and refer to them simply as interpretations.

Definition 2 (The CHC-SAT Problem). *Given a set of CHCs Π, determine whether Π is satisfiable.*

Note that, if Π is *unsatisfiable*, then there exists a refutation (a proof of unsatisfiability) in the form of a ground derivation of $\bot$ [8]. Along with determining whether Π is satisfiable, we are often interested in finding a solution to it. A *solution* for a set of CHCs Π is either a satisfying interpretation, when Π is satisfiable, or a ground refutation, when Π is unsatisfiable.

Finally, for a FOL formula ψ, we denote by $\mathcal{P}^{\psi}$ the set of all uninterpreted predicate symbols that occur in ψ. Given a set of FOL formulas Ψ, $\mathcal{P}^{\Psi}$ denotes the set $\bigcup_{\psi \in \Psi} \mathcal{P}^{\psi}$. Note that an interpretation for Π is defined over $\mathcal{P}^{\Pi}$.

Example 1. As an example, consider the following schematic set of CHCs over the set $\{p_1, p_2, p_3, p_4\}$ of uninterpreted predicate symbols:

$$\top \rightarrow p_1(x) \tag{1}$$
$$p_1(x) \wedge \varphi_2(x, y) \rightarrow p_2(x, y) \tag{2}$$
$$p_1(x) \wedge \varphi_3(x, z) \rightarrow p_3(z) \tag{3}$$
$$p_1(x) \wedge p_1(y) \wedge p_3(z) \wedge \varphi_4(x, y, z) \rightarrow p_4(x, y) \tag{4}$$
$$p_4(x, y) \wedge \varphi_5(x, y, z) \rightarrow p_2(x, z) \tag{5}$$
$$p_2(x, z) \wedge \varphi_6(x, z) \rightarrow \bot \tag{6}$$

It consists of 5 rules (Clauses 1–5) and a query (Clause 6). Clause 1 is a fact and Clause 4 is nonlinear, since it includes more than one predicate symbol in its body. Note that, the predicate symbol p_1 occurs twice in the body of Clause 4. However, $\mathcal{P}^{body(4)}$ is the set $\{p_1, p_3\}$ (rather than a multiset), where repetitions are ignored.

2.2 Hypergraphs and Hyperpaths

The definitions in this subsection resemble [1]. A *directed hypergraph* $G = (V, E)$ consists of a nonempty set of nodes V and a set of hyperedges E. A *hyperedge* connects several source nodes to a single target node. It is represented by a pair $e = (S, t)$, where $S \subseteq V$ is the (nonempty) set of source nodes of e, denoted $source(e)$ and $t \in V$ is the target node of e, denoted $target(e)$.

Definition 3 (Nontrivial Hyperpath). *A* nontrivial hyperpath *in $G = (V, E)$ from a set of sources $S \subseteq V$ to a target $t \in V$ is a nonempty set of hyperedges $E_{S,t} \subseteq E$ with the following property: the hyperedges in $E_{S,t}$ can be ordered as a vector $(e_1, \ldots, e_k)$ where,*

1. *$source(e_i) \subseteq (S \cup \{target(e_1), \ldots, target(e_{i-1})\})$ for every e_i in $E_{S,t}$.*
2. *$target(e_k) = t$.*
3. *There is no nonempty $E' \subset E_{S,t}$ that satisfies 1 and 2.*

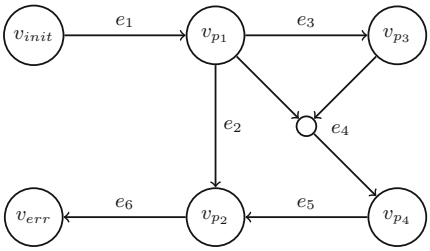

Fig. 1. A hypergraph example.

According to the above definition, a hyperpath must include at least one hyperedge. We therefore refer to such hyperpaths as nontrivial. Note that a hyperpath with an empty set of hyperedges (*trivial hyperpath*) is possible in [1], however, in our context we do not consider such hyperpaths. In the sequel, hyperpaths are always nontrivial.

Let $E_{S,t}$ be a hyperpath form S to t. Due to the minimality of a hyperpath (condition 3 above), it holds that for every two different hyperedges $e_i, e_j \in E_{S,t}$, $target(e_i) \neq target(e_j)$. This and Definition 3 imply the following property.

Property 1. Let $E_{S,t}$ be a hyperpath form S to t and let (X, t) be the unique hyperedge in $E_{S,t}$ leading to t. Then, $E_{S,t}$ can be written as follows:

$$E_{S,t} = \{(X,t)\} \cup (\bigcup_{x \in (X \setminus S)} E_{S,x}),$$

where for every $x \in (X \setminus S)$, $E_{S,x}$ is the hyperpath from S to x included in $E_{S,t}$.

Next, we add weights to the hyperedges of a hypergraph $G = (V, E)$. This is done with a *weight function* $w : E \to \mathbb{N}$, which associates a non-negative integer with each hyperedge. A weight function w for hyperedges can be lifted to a weight function $\hat{w}$ for hyperpaths, as follows.

Definition 4 (Weight Function for Hyperpaths). *Let $E_{S,t}$ be a hyperpath from S to t and let (X, t) be the unique hyperedge in $E_{S,t}$ leading to t. According to Property 1, $E_{S,t} = \{(X,t)\} \cup (\bigcup_{x \in (X \setminus S)} E_{S,x})$. The* weight function $\hat{w}$ *for $E_{S,t}$ is defined inductively in the following way:*

$$\hat{w}(E_{S,t}) = w((X,t)) + \sum_{x \in (X \setminus S)} \hat{w}(E_{S,x})$$

That is, the weight of a hyperpath is defined as the sum of the weight of its last hyperedge with the weights of the hyperpaths leading to each of its sources.[1]

[1] This weight function is called the traversal cost in [1]. For this weight function, computing the shortest nontrivial hyperpath (see Sect. 4) is polynomial.

Example 2. Consider the hypergraph in Fig. 1 and the hyperpath $E_{\{v_{init}\},v_{p_4}} = \{e_1, e_3, e_4\}$. Assume the weight function for each edge is the number of its sources: $w(e_i) = 1$ for each $i \neq 4$ and $w(e_4) = 2$. The weight of this hyperpath is

$$\begin{aligned}\hat{w}(E_{\{v_{init}\},v_{p_4}}) &= w(e_4) + \hat{w}(E_{\{v_{init}\},v_{p_1}}) + \hat{w}(E_{\{v_{init}\},v_{p_3}}) \\ &= w(e_4) + \hat{w}(E_{\{v_{init}\},v_{p_1}}) + (w(e_3) + \hat{w}(E_{\{v_{init}\},v_{p_1}})) \\ &= w(e_4) + w(e_1) + (w(e_3) + w(e_1)) = 2 + 1 + (1 + 1) = 5.\end{aligned}$$

Notice that the weight of e_1 is considered twice in the weight of $E_{\{v_{init}\},v_{p_4}}$ since e_1 belongs to both $E_{\{v_{init}\},v_{p_1}} = \{e_1\}$ and $E_{\{v_{init}\},v_{p_3}} = \{e_1, e_3\}$.

3 Structure-Guided, Lazy and Incremental CHC Solving

In this section, we present StHorn - a structure-guided, lazy and incremental technique for CHC-solving. Given a set Π of CHCs, StHorn constructs a solution for Π by iteratively examining a monotone sequence of its subsets. It starts by choosing a subset of clauses $\Delta \subseteq \Pi$, and iteratively adds clauses to it, as needed. If at any point, the subset becomes unsatisfiable, StHorn halts and returns UNSAT. Otherwise, if a subset of clauses is satisfiable, StHorn tries to extend the satisfying interpretation for the subset into an interpretation for Π in an incremental fashion. To this end, StHorn maintains an interpretation $\mathcal{I}$ that is injected into the CHC-solver as a starting point at each iteration, enabling the solver to search for a solution within a reduced state space. In order for the solver to return a sound solution when it is invoked to solve the current Δ, $\mathcal{I}$ must satisfy all rules in Δ. The following definition captures this requirement.

Definition 5 (Rule-Satisfiability). *Let Δ be a set of CHCs, and $\mathcal{I}$ be an interpretation.* $\mathcal{I}$ rule-satisfies Δ, *denoted $\mathcal{I} \models_r \Delta$, if $\mathcal{I} \models \pi$ for every rule $\pi \in \Delta$. Note that in this case, $\mathcal{I}$ may not satisfy some of the queries in Δ.*

In what follows, we assume that the underlying used CHC-solver can receive a set Δ of CHCs and an initial interpretation $\mathcal{I}$ for the predicates in $\mathcal{P}^{\Delta}$. Further, we assume that the solver returns a sound solution whenever $\mathcal{I} \models_r \Delta$. We leave the discussion on this assumption to the end of the section.

In Sect. 6.1 we show that, in fact, the StHorn technique can be implemented on top of any existing CHC-solver. This includes solvers that cannot receive initial interpretations for the predicates.

We start with a simple, high-level description of the technique. In the following sections, we go into the fine-grained details of the implementation. The pseudo-code of StHorn appears in Algorithm 1. The specifications of the algorithm and of its internal procedures are summarized in Fig. 2.

StHorn receives a set Π of CHCs. As mentioned, it maintains a subset of clauses $\Delta \subseteq \Pi$ and an interpretation $\mathcal{I}$ that rule-satisfies Δ. StHorn starts by calling `Select` (line 1) and initializing Δ to be some subset of Π (i.e., $\Delta \subseteq \Pi$). Next, it initializes the interpretation $\mathcal{I}$ of every uninterpreted predicate that

Algorithm 1. StHorn(Π)

Input: A set Π of CHCs
Output: A solution to Π

```
1  Δ ← Select(Π, ∅)
2  I(p) ← ⊤, ∀p ∈ P^Δ
3  while ⊤ do
4      (res, I', R) ← Solve(Δ, I)
5      if res = UNSAT then
6          return (UNSAT, −, R)
7      else                                  ▷ i.e., res = SAT
8          if Δ = Π then
9              return (SAT, I', −)
10         else                              ▷ i.e., Δ ⊂ Π
11             δ ← Select(Π, Δ)
12             Δ ← Δ ∪ δ
13             I ← Amend(I', Δ, δ)
14         end if
15     end if
16 end while
```

occurs in Δ to $\top$ (line 2). Note that after initialization, $\mathcal{I} \models_r \Delta$ (see the proof of Theorem 1).

StHorn then moves to the main loop (line 3). Every iteration begins by checking the satisfiability of Δ by invoking the underlying CHC-solver with a call to `Solve` (line 4). Consider the case in which `Solve` returns UNSAT and a ground refutation $\mathcal{R}$ for Δ. Since Δ consists entirely of clauses from Π, $\mathcal{R}$ is also a refutation for Π. Thus, StHorn returns UNSAT and $\mathcal{R}$ (line 6). Now, consider the case in which `Solve` returns SAT and a satisfying interpretation $\mathcal{I}'$ for Δ. If Δ is equal to Π, StHorn returns SAT and $\mathcal{I}'$ as the satisfying interpretation of Π (line 9). Otherwise, Δ is a strict subset of Π. As a preparation for the next iteration, Δ is extended and the interpretation is amended accordingly. First, the method `Select` selects a set of fresh clauses δ from $\Pi \setminus \Delta$ (line 11) that are added to Δ (line 12). We require that at least one clause is selected (i.e., $\delta \neq \emptyset$) to guarantee progress. At this stage, $\mathcal{I}'$ may no longer be a rule-satisfying interpretation with respect to the extended Δ. As a remedy, StHorn invokes `Amend` (line 13), which modifies $\mathcal{I}'$ in order to make it rule-satisfying for Δ before the subsequent call to `Solve`.[2]

Remark 1 (Termination of StHorn). In general, the CHC-SAT problem is undecidable, so termination is not guaranteed. However, if every call to `Solve` made by StHorn terminates, then StHorn terminates as well. The reason for this is the requirement that `Select` must always return at least one fresh clause from Π.

[2] The underlying CHC-solver may also return UNKNOWN. This case can be handled similarly to the case where SAT is returned and is omitted for simplicity of presentation.

$(res, \mathcal{I}', \mathcal{R}) \leftarrow \mathsf{StHorn}(\Pi)$
Requires: $\top$
Ensures: $(res = \text{SAT} \Rightarrow \mathcal{I}' \models \Pi)$ **and**
$(res = \text{UNSAT} \Rightarrow \mathcal{R}$ is a ground refutation of $\Pi)$

$(res, \mathcal{I}', \mathcal{R}) \leftarrow \texttt{Solve}(\Delta, \mathcal{I})$
Requires: $\mathcal{I} \models_r \Delta$
Ensures: $(res = \text{SAT} \Rightarrow \mathcal{I}' \models \Delta)$ **and**
$(res = \text{UNSAT} \Rightarrow \mathcal{R}$ is a ground refutation of $\Delta)$

$\delta \leftarrow \texttt{Select}(\Pi, \Delta)$
Requires: $\Delta \subseteq \Pi$
Ensures: $\delta \subseteq \Pi \setminus \Delta$ **and** $(\Delta \subset \Pi \Rightarrow \delta \neq \emptyset)$

$\mathcal{I} \leftarrow \texttt{Amend}(\mathcal{I}', \Delta, \delta)$
Requires: $\delta \subseteq \Delta$ **and** $\mathcal{I}' \models \Delta \setminus \delta$
Ensures: $\mathcal{I} \models_r \Delta$

Fig. 2. Specifications for the StHorn Algorithm and its Procedures

Theorem 1 (Correctness of StHorn). *Let Π be a set of CHCs given to StHorn. If StHorn returns SAT (UNSAT), then Π is satisfiable (unsatisfiable).*

Proof. First, we show that at every call to `Solve`, $\mathcal{I} \models_r \Delta$. When `Solve` is called for the first time, following the initialization of $\mathcal{I}$, it holds that $\mathcal{I}(p) = \top$ for all $p \in \mathcal{P}^{\Delta}$. Let $\pi := p_1 \wedge \cdots \wedge p_k \wedge \varphi \rightarrow q$ be a rule in Δ. The formula $\mathcal{I}(q)$, which is $\top$, is implied by any other formula, and in particular, it is implied by $\mathcal{I}(p_1) \wedge \cdots \wedge \mathcal{I}(p_k) \wedge \varphi$. Therefore, $\mathcal{I}(\pi)$ is valid, i.e., $\mathcal{I} \models \pi$. Accordingly, $\mathcal{I} \models_r \Delta$. In later iterations, `Solve` is called after `Amend`, which ensures $\mathcal{I} \models_r \Delta$ as well. Therefore, the requirement in the specifications of `Solve` holds in every invocation.

Assume StHorn returns SAT. From the definition of the algorithm, it follows that `Solve` was invoked at the last iteration with Π, and that it returned SAT and $\mathcal{I}'$. By the specifications of `Solve`, it is guaranteed that Π is indeed satisfiable and that $\mathcal{I}'$, which is returned by StHorn, satisfies Π. Finally, assume StHorn returns UNSAT. From the definition of StHorn, it follows that `Solve` was invoked at the last iteration with a subset $\Delta \subseteq \Pi$, and that it returned UNSAT and $\mathcal{R}$. By the specifications of `Solve`, it is guaranteed that Δ is indeed unsatisfiable and that $\mathcal{R}$, which is returned by StHorn, is a ground refutation for Δ. Since Δ consists entirely of clauses from Π, $\mathcal{R}$ is also a ground refutation for Π. □

Requiring Rule-Satisfiablity. We require the CHC-solver to return a sound solution, given a set of CHCs and a rule-satisfying interpretation. This is essential, since if there exists a rule that is not satisfied by the injected interpretation, the solver may return an incorrect result. As an example, consider the following unsatisfiable set Π of CHCs: $\{x = 0 \rightarrow p(x),\ p(x) \wedge x = 0 \rightarrow \bot\}$. When given an

interpretation that is not rule-satisfying, such as the one that maps $p(x)$ to $\bot$, the CHC-solver might conclude that Π is satisfiable after examining the query and observing that it is satisfied by the injected interpretation.

Rule-satisfiablity is also important in that, whenever a set of CHCs is satisfiable, any rule-satisfying interpretation for it can be strengthened into a satisfying one.[3] For example, consider the following satisfiable set Π' of CHCs: $\{x = 0 \to p(x),\ p(x) \wedge x \neq 0 \to \bot\}$. Π' is clearly satisfied by the interpretation that maps $p(x)$ to the formula $x = 0$. Now, consider the interpretation that maps $p(x)$ to $x \geq 0$. While this rule-satisfying interpretation does not satisfy Π', as it does not satisfy its query, it can be strengthened by the solver into the above satisfying interpretation. Note also, that supplying the solver with this initial interpretation narrows its search space, as otherwise it would have began with the interpretation that maps $p(x)$ to $\top$.

4 Structure-Guided Selection of CHCs

Recall that a CHC is of the form $p_1 \wedge \cdots \wedge p_k \wedge \varphi \to q$ (the variable vectors are omitted for readability). For brevity, we denote such a clause by the triple $\langle \{p_1, \ldots, p_k\}, \varphi, q \rangle$. Similarly, a fact and a query are denoted by $\langle \emptyset, \varphi, q \rangle$ and $\langle \{p_1, \ldots, p_k\}, \varphi, \bot \rangle$, respectively.[4] It should be noted that a clause may contain several occurrences of the same predicate symbol in its body (see, for example, rule 4 of Example 1). For the purpose of guiding the CHC selection, it is sufficient to refer to the predicates appearing in the clause body as a set rather than a multiset. That is, repetitions of predicate symbols in the same body are ignored. In what follows, Π is the given set of CHCs.

There are two key ingredients that affect the efficiency of StHorn: (1) the choice of clauses to be examined at each iteration; and (2) the incremental construction of the interpretation. The first task is performed by the procedure `Select`, which we describe in this section. The second task is performed by the underlying CHC-solver and the procedure `Amend`, which we describe in Sect. 5.

For capturing the structure of the problem, it is useful to model Π as a directed hypergraph with parallel edges, whose vertices and hyperedges represent the predicate symbols and clauses, respectively.

Definition 6 (Induced CHC Hypergraph). *Let Π be a set of CHCs. The* induced CHC hypergraph *of Π is a directed hypergraph $G_\Pi = (V_\Pi, E_\Pi)$, where*

$$
\begin{aligned}
V_\Pi &= \{v_p \mid p \in \mathcal{P}^\Pi\} \cup \{v_{init}, v_{err}\} \\
E_\Pi &= \{(\{v_{init}\}, v_q) \mid \langle \emptyset, \varphi, q \rangle \in \Pi\} \cup \\
&\quad \{(\{v_{p_1}, \ldots, v_{p_k}\}, v_q) \mid \langle \{p_1, \ldots, p_k\}, \varphi, q \rangle \in \Pi\} \cup \\
&\quad \{(\{v_{p_1}, \ldots, v_{p_k}\}, v_{err}) \mid \langle \{p_1, \ldots, p_k\}, \varphi, \bot \rangle \in \Pi\}
\end{aligned}
$$

[3] Strengthening can be achieved as follows: if $\mathcal{I} \models_r \Pi$ and $\mathcal{I}' \models \Pi$ then the interpretation that maps every $p \in \mathcal{P}^\Pi$ to $\mathcal{I}(p) \wedge \mathcal{I}'(p)$ satisfies Π.

[4] We assume, w.l.o.g., that the head of a query is $\bot$.

There is a correspondence between the hypergraph (vertices and hyperedges) and the set of CHCs (predicates and clauses). More precisely, a vertex $v_p \in V_\Pi$ corresponds to the predicate $p \in \mathcal{P}^\Pi$, and the vertices v_{init} and v_{err} correspond to $\top$ and $\bot$, respectively. Also, an edge $e_\pi \in E_\Pi$ corresponds to the clause $\pi \in \Pi$.[5] We assume that all the edges of the CHC hypergraph are on a hyperpath from v_{init} to v_{err} (see Definition 3). Otherwise, such edges can be removed from the graph without changing the solution of Π.

Example 3. The hypergraph depicted in Fig. 1 is exactly the induced CHC hypergraph for the set of clauses of Example 1. Note that, Clause 4 is represented by the hyperedge e_4, whose set of sources is $\{v_{p_1}, v_{p_3}\}$. As mentioned above, for our algorithms, there is no need to remember the repetitions of the predicate symbol p_1 in the body of Clause 4.

The procedure `Select` is iteratively called by StHorn in order to select the subsets $\Delta \subseteq \Pi$ to be examined. For this purpose, it explores paths in the induced CHC hypergraph. Our approach is aimed at finding a solution to Π lazily and incrementally, so `Select` chooses small subsets of clauses that correspond to shortest nontrivial hyperpaths in the graph. The proposed selection strategy is based on the understanding that solving small subsets is often easier and can be advantageous to the overall solution.

Recall that there is no restriction on the selected subsets, except that they must include at least one fresh clause from Π to guarantee progress. Nevertheless, we further require `Select` to produce only subsets in which all clauses are on a hyperpath from v_{init} to v_{err}. Otherwise, if there exists a node v_p which is not reachable from v_{init}, then there exists a trivial interpretation that assigns $\bot$ to p. Similarly, if v_{err} is not reachable from v_p, then there exists a trivial interpretation that assigns $\top$ to p.

We start by presenting the algorithm `ShortNt` for finding the shortest nontrivial hyperpath from a set of sources U to each node in the graph. This algorithm is a modification of the algorithm presented in [3,4], for finding the shortest hyperpath in a directed, weighted hypergraph, from a given node to each of the nodes in the graph. Our algorithm is different from the above in two ways. First, the shortest path starts at a given *set of source nodes* U. Second, we search for only *nontrivial hyperpaths*. That is, hyperpaths that consist of at least one hyperedge.

Algorithm `ShortNt`, depicted in Algorithm 2, gets as input a hypergraph $G = (V, E)$, a weight function $w : E \to \mathbb{N}$ and a source set $U \subseteq V$. It returns a map $Dist : V \to \mathbb{N} \cup \{\infty\}$, which associates with each node v the weight of the shortest, nontrivial hyperpath $E_{U,v}$ from U to v (i.e., $\hat{w}(E_{U,v})$). It also returns a map $Last : V \to E \cup \{\texttt{null}\}$, which associates with each node v, the hyperedge e on $E_{U,v}$ for which $target(e) = v$. If v is not reachable from U along a nontrivial

[5] In fact, the induced CHC hypergraph may include parallel hyperedges originating from two CHCs that differ only in their constraints. While we support such a case, we omit it here for simplicity of presentation.

Algorithm 2. `ShortNt`(G, w, U)

Input: A hypergraph $G = (V, E)$, a hyperedge weight function $w : E \to \mathbb{N}$ and a source set $U \subseteq V$

Output: A map $Dist : V \to \mathbb{N} \cup \{\infty\}$ and a map $Last : V \to E \cup \{\texttt{null}\}$

```
1  Count(e) ← |S|, ∀e = (S,t) ∈ E
2  Dist(v) ← ∞, ∀v ∈ V
3  Last(v) ← null, ∀v ∈ V
4  Q ← ∅
5  for all v ∈ U do
6      Visit(v)
7  end for
8  while Q ≠ ∅ do
9      v ← arg min_{u∈Q} Dist(u)
10     Q ← Q \ {v}
11     Visit(v)
12 end while
13 return (Dist, Last)

Visit(v)
14 for all e = (S,t) ∈ E s.t. v ∈ S do
15     Count(e) ← Count(e) − 1
16     if Count(e) = 0 then
17         D ← w(e) + Σ_{v∈(S\U)} Dist(v)
18         if D < Dist(t) then
19             Dist(t) ← D
20             Last(t) ← e
21             Q ← Q ∪ {t}
22         end if
23     end if
24 end for
```

hyperpath, then the values $Dist(v) = \infty$ and $Last(v) = \texttt{null}$ are returned. Initially, $Dist(v) = \infty$ and $Last(v) = \texttt{null}$, for every $v \in V$ (lines 2, 3).

In addition to $Dist$ and $Last$, `ShortNt` maintains a map $Count : E \to \mathbb{N}$, such that for each hyperedge e, $Count(e)$ is the number of sources of e that have not been visited so far. $Count(e)$ is initialized to $|source(e)|$ (line 1). It is decremented by 1 whenever a source node of e is visited (line 15). Only when it is set to 0, the hyperedge e is processed (lines 16–21). `ShortNT` also maintains a set $\mathcal{Q}$, which contains the nodes in V that are yet to be processed.

`ShortNT` first processes all source nodes $v \in U$ (lines 5–7). It goes over all edges e for which v is a source (line 14) and decrements $Count(e)$. If $Count(e)$ is now 0, meaning that all its sources have already been visited, then $Dist(target(e))$ and $Last(target(e))$ are updated. This is done when a shorter hyperpath to $target(e)$, containing e, is found. In this case, $target(e)$ is added to $\mathcal{Q}$ (lines 16–21). As long as $\mathcal{Q}$ is not empty, a node v with minimal $Dist(v)$ is removed from $\mathcal{Q}$ and is processed (lines 8–11).

Remark 2 (Complexity of `ShortNt`*).* By a similar argument to the correctness proof of Dijkstra's shortest path algorithm, we can show that `ShortNt` inserts every node to $\mathcal{Q}$ and processes it at most once. Consequently, the algorithm is polynomial in the size of the hypergraph.

Lemma 1 (Correctness of *`ShortNT`*). *Given a graph $G = (V, E)$, a weight function w and a source set U, then for every node $v \in V$ reachable from U, `ShortNT` returns $Dist(v)$ and $Last(v)$ so that $Dist(v)$ is the weight of the shortest, nontrivial hyperpath $E_{U,v}$ from U to v, and $Last(v)$ is the edge e in $E_{U,v}$ such that $target(e) = v$.*

Algorithm 3. `Select`(Π, Δ)

Input: A set Π of CHCs and a subset $\Delta \subseteq \Pi$
Output: A subset $\delta \subseteq \Pi \setminus \Delta$ such that $\delta \neq \emptyset$ if $\Delta \subset \Pi$

```
1  let G_{Π\Δ} = (V_{Π\Δ}, E_{Π\Δ})
2  Reach ← {v_p ∈ V_{Π\Δ} | p ∈ P^Δ}
3  w(e) ← |S|, ∀e = (S, t) ∈ E_{Π\Δ}
4  (Dist, Last) ← ShortNt(G_{Π\Δ}, w, {v_init} ∪ Reach)
5  v ← arg min_{u∈Reach∪{v_err}} Dist(u)
6  Opt ← {v}
7  δ ← ∅
8  while Opt ≠ ∅ do
9      let u ∈ Opt
10     Opt ← Opt \ {u}
11     if u ∉ ({v_init} ∪ Reach) then
12         e ← Last(u)
13         δ ← δ ∪ {π(e)}
14         Opt ← Opt ∪ source(e)
15     end if
16 end while
17 return δ
```

Next, we describe the procedure `Select`, given in Algorithm 3. It gets as input a subset of clauses $\Delta \subseteq \Pi$ and explores the graph $G_{\Pi \setminus \Delta}$. It returns a set $\delta \subseteq \Pi \setminus \Delta$, which is nonempty if Δ is a strict subset of Π. `Select` starts by initializing a set of nodes *Reach*, which consists of all nodes in $G_{\Pi \setminus \Delta}$, corresponding to predicate symbols that appear in Δ (line 2). Next, it sets the weight of each hyperedge to the number of its sources (line 3). It now computes *Dist* and *Last* by calling `ShortNt` on the graph $G_{\Pi \setminus \Delta}$, with weights w as defined above, and the set of sources $\{v_{init}\} \cup Reach$ (line 4). Note that, any node in the graph originating from the previously processed set Δ is now a source for `ShortNt`.

From all shortest paths computed by `ShortNt`, `Select` chooses the shortest among those whose final target is a node v in either *Reach* or $\{v_{err}\}$ (line 5). Thus, the chosen path $E_{H,v}$ starts at $H = \{v_{init}\} \cup Reach$ and ends in $v \in (\{v_{err}\} \cup Reach)$. In lines 6–16, the chosen path is traversed backwards from v, producing the set of hyperedges on it and accumulating their corresponding clauses in δ (line 13). Note that, $\pi(e)$ in line 13 returns the clause corresponding to the hyperedge e.

Lemma 2 (Correctness of `Select`). *Given a set Π of CHCs and a subset $\Delta \subseteq \Pi$,* `Select` *returns a subset $\delta \subseteq \Pi \setminus \Delta$, which is nonempty if $\Delta \subset \Pi$.*

5 Ensuring Rule-Satisfiability

In this section, we describe the procedure `Amend`. First, we describe a simplified version of the procedure, and then present two modifications that can enhance its performance (Sects. 5.1 and 5.2).

Algorithm 4. `Amend`$(\mathcal{I}', \Delta, \delta)$

Input: A set Δ of CHCs, a subset $\delta \subseteq \Delta$ and an interpretation $\mathcal{I}'$ that satisfies $\Delta \setminus \delta$
Output: An interpretation $\mathcal{I}$ that rule-satisfies Δ

```
1  I(p) ← I'(p), ∀p ∈ P^{Δ\δ}
2  I(p) ← ⊤, ∀p ∈ (P^δ \ (P^{Δ\δ}))
3  Q ← {π ∈ δ | head(π) ≠ ⊥}
4  while Q ≠ ∅ do
5      let π = ⟨X, φ, q⟩ ∈ Q
6      Q ← Q \ {π}
7      (res, −, −) ← Solve({I(π)}, −)
8      if res = UNSAT then
9          I(q) ← ⊤
10         Q ← Q ∪ {π' ∈ Δ | q ∈ P^{body(π')} ∧ head(π') ≠ ⊥}
11     end if
12 end while
```

Consider the StHorn algorithm again. At line 12, a subset δ of new clauses from Π is added to Δ. At this point, it is no longer guaranteed that the current interpretation $\mathcal{I}'$ rule-satisfies Δ. In order to maintain the correctness of StHorn, $\mathcal{I}'$ must be modified before the next call to `Solve`. The modification of $\mathcal{I}'$ is performed by the procedure `Amend`. The goal of `Amend` is to construct an interpretation $\mathcal{I}$ such that $\mathcal{I} \models_r \Delta$, while preserving as many parts as possible from the existing interpretation $\mathcal{I}'$. This makes StHorn incremental when invoking `Solve`, as it allows the CHC-solver to use previously learned information that narrows the state space. In the worst case, predicates in $\mathcal{I}$ are reset back to $\top$.

The pseudo-code of the procedure appears in Algorithm 4. `Amend` is given a set Δ of CHCs, a subset $\delta \subseteq \Delta$, and an interpretation $\mathcal{I}'$ that satisfies all clauses in Δ, except possibly the clauses in δ. The procedure constructs and returns an interpretation $\mathcal{I}$ that rule-satisfies Δ. First, the interpretation of all predicates that occur in the previous examined subset ($\Delta \setminus \delta$) is initialized to the current interpretation $\mathcal{I}'$ (line 1) and the interpretation of all *fresh* predicates (i.e., predicates that occur in δ but not in $\Delta \setminus \delta$) is initialized to $\top$ (line 2). The procedure maintains a set of clauses $\mathcal{Q}$, consisting of all rules in Δ that might not be satisfied by $\mathcal{I}$. According to the specifications of the procedure, those clauses are initially the rules in δ, so $\mathcal{Q}$ is initialized accordingly (line 3).

`Amend` then proceeds to its main loop (line 4). At each iteration, a clause $\pi = \langle X, \varphi, q \rangle$ is removed from $\mathcal{Q}$ (lines 5–6). Then, in order to check whether $\mathcal{I} \models \pi$, a new CHC-SAT problem consisting of a single clause $\mathcal{I}(\pi)$ is constructed and sent to `Solve` (line 7). As $\mathcal{I}(\pi)$ does not contain any uninterpreted predicate symbol, no initial interpretation is injected into the solver. If $\mathcal{I} \models \pi$, then nothing has to be done and a new iteration begins. Otherwise, the interpretation of the head predicate q is reset to $\top$ (line 9). After weakening the interpretation of q, $\mathcal{I}$ may no longer satisfy all rules in Δ where q is one of the body predicates. Therefore, any such rule is added to $\mathcal{Q}$ (line 10), and the forward amendment process continues.

Remark 3 **(Termination of `Amend`).** During the execution of `Amend`, a rule in Δ is added to $\mathcal{Q}$ only if the interpretation of one of its body predicates is reset to $\top$ (lines 9–10). For every predicate $q \in \mathcal{P}^{\Delta}$, such a reset can occur at most once, since afterward every rule with q as the head predicate is satisfied trivially. Due to the above, and since every rule has a finite number of body predicates, a rule is inserted into $\mathcal{Q}$ finitely many times. Therefore, if every call to `Solve` during the execution of `Amend` terminates, then `Amend` terminates as well.

Lemma 3 (Correctness of `Amend`). *Let Δ be a subset of CHCs, δ be a subset of Δ, and $\mathcal{I}'$ an interpretation that satisfies $\Delta \setminus \delta$. Then, if* `Amend` *terminates on Δ, δ and $\mathcal{I}'$, it returns an interpretation $\mathcal{I}$ that rule-satisfies Δ.*

Proof. Let $\pi = p_1 \wedge \cdots \wedge p_k \wedge \varphi \rightarrow q$ be a rule in Δ. Let n be the number of iterations that the main loop of `Amend` was executed and ℓ be the last iteration in which π was removed from $\mathcal{Q}$. We denote by $\mathcal{I}_j$ the interpretation after the j-th iteration of the loop. We will show that $\mathcal{I}_n \models \pi$, i.e., that $\mathcal{I}_n(p_1) \wedge \cdots \wedge \mathcal{I}_n(p_k) \wedge \varphi \Rightarrow \mathcal{I}_n(q)$.

Consider the case in which $\ell = 0$. In this case, π was never added to $\mathcal{Q}$ during the execution of `Amend`. First, we claim that $\mathcal{I}_n(p_i) = \mathcal{I}_0(p_i)$ for $1 \leq i \leq k$. This holds, because, if there existed an iteration in which the interpretation of some p_i was changed (line 9), then π would have been added to $\mathcal{Q}$ (line 10). Moreover, by the initialization of $\mathcal{I}$ and $\mathcal{Q}$ (lines 1 and 3), we have that $\mathcal{I}_0$ agrees with $\mathcal{I}'$ on all predicates in $\mathcal{P}^{\Delta \setminus \delta}$ and that $\pi \in \Delta \setminus \delta$. Therefore, since it is required that $\mathcal{I}' \models \Delta \setminus \delta$, it holds that $\mathcal{I}_0 \models \pi$. Finally, because the interpretation of every predicate may only be weakened in `Amend`, for every $j_1 < j_2$ and $r \in \mathcal{P}^{\Delta}$ it holds that $\mathcal{I}_{j_1}(r) \Rightarrow \mathcal{I}_{j_2}(r)$. Therefore, $\mathcal{I}_0(q)$ implies $\mathcal{I}_n(q)$. To summarize, we have:

$$\mathcal{I}_n(p_1) \wedge \cdots \wedge \mathcal{I}_n(p_k) \wedge \varphi \equiv \mathcal{I}_0(p_1) \wedge \cdots \wedge \mathcal{I}_0(p_k) \wedge \varphi \Rightarrow \mathcal{I}_0(q) \Rightarrow \mathcal{I}_n(q)$$

Now, consider the case in which $\ell > 0$. Similarly, we establish the following:

$$\mathcal{I}_n(p_1) \wedge \cdots \wedge \mathcal{I}_n(p_k) \wedge \varphi \equiv \mathcal{I}_\ell(p_1) \wedge \cdots \wedge \mathcal{I}_\ell(p_k) \wedge \varphi \Rightarrow \mathcal{I}_\ell(q) \Rightarrow \mathcal{I}_n(q)$$

Here, the first implication holds since when a CHC is removed from $\mathcal{Q}$, it is either satisfied by the current interpretation, or the interpretation of its head predicate is set to true. Thus, $\mathcal{I}_n \models \pi$ as needed. □

In the remainder of the section, we describe two modifications to `Amend` aimed at extracting and preserving more information from the amended interpretation.

5.1 Exploiting Conjunctive Interpretations

The first modification to `Amend` exploits the shape of the interpretation formulas. Many solvers operate on formulas in the form $c_1 \wedge \cdots \wedge c_n$ (e.g. Conjunctive Normal Form). Recall that `Amend` checks whether $\mathcal{I} \models_r \Delta$ after the addition of new clauses from Π. For every checked rule $\pi = p_1 \wedge \cdots \wedge p_k \wedge \varphi \rightarrow q$, it is checked whether $\mathcal{I}(p_1) \wedge \cdots \wedge \mathcal{I}(p_k) \wedge \varphi \rightarrow \mathcal{I}(q)$ is valid. If $\mathcal{I}(q)$ is not implied by

$\mathcal{I}(p_1) \wedge \cdots \wedge \mathcal{I}(p_k) \wedge \varphi$, then $\mathcal{I}(q)$ is reset to $\top$. After this update to $\mathcal{I}$, it holds that $\mathcal{I} \models \pi$. However, all the information previously learnt regarding q is lost. When $\mathcal{I}(q)$ is a conjunction formula $c_1 \wedge \cdots \wedge c_n$, we can check each conjunct separately, i.e., for every $1 \leq i \leq n$ we check whether $\mathcal{I}(p_1) \wedge \cdots \wedge \mathcal{I}(p_k) \wedge \varphi \rightarrow c_i$ is valid. Then, we remove from $\mathcal{I}(q)$ only the conjuncts c_i that are not implied. In the worst case, no conjuncts are implied, and $\mathcal{I}(q)$ is reset to $\top$. In practice, we can often retain significant parts of the interpretation using this approach. After applying this optimization, rules in Δ might be inserted into $\mathcal{Q}$ additional times. Nevertheless, since every conjunctive interpretation has a finite number of conjuncts, each rule is still inserted into $\mathcal{Q}$ a finite number of times.

5.2 Extending Existing Interpretations

In this subsection, we introduce a preliminary step that, if successful, will eliminate the need to run `Amend`. Before amending the interpretation, one can try to extend $\mathcal{I}'$ for the fresh predicates (i.e., predicates in $\mathcal{P}^{\delta} \setminus (\mathcal{P}^{\Delta \setminus \delta})$). For this, we construct a new CHC-SAT problem with the following set of CHCs: $\delta' = \{\mathcal{I}'(\pi) \mid \pi \in \delta \wedge \mathit{head}(\pi) \neq \bot\}$. δ' is created by substituting every non-fresh predicate (i.e., every predicate in $\mathcal{P}^{\Delta \setminus \delta}$) with its $\mathcal{I}'$ interpretation in every rule in δ. All fresh predicates remain uninterpreted.

When `Amend` is invoked, it first constructs δ' and calls `Solve`. If δ' is satisfiable, $\mathcal{I}'$ is extended for the fresh predicates according to the satisfying interpretation returned by `Solve`. In this case, `Amend` halts and returns the new interpretation without further checks. Otherwise, if δ' is unsatisfiable, `Amend` is executed as before.

6 Implementation Details and Experimental Evaluation

6.1 Implementation Details

We implemented StHorn as an open-source generic framework in C++. In addition, we implemented two instances of StHorn: one using Spacer [28] through the C++ API of Z3's [32]. The other uses Eldarica [24] as a CHC-solver. For the Eldarica instance we implemented a JAVA API for Eldarica (which is implemented in Scala). Then, we used JNI in order to invoke Eldarica (through the JAVA API we implemented) from our C++ framework. Our implementation is available in https://github.com/omerap/StructuralHorn.

StHorn with Spacer*:* We denote this instance of StHorn as StHorn_S. Spacer is based on IC3/PDR [9,23,28]. Satisfying interpretations are given in Conjunctive Normal Form (CNF), and the Z3 API allows to pre-load interpretations for the predicates appearing in the CHCs. This is done by adding conjuncts to a given predicate. Adding "partial" interpretations to the predicates allowed us to use Spacer incrementally seamlessly, without modifying the set of CHCs.

StHorn with Eldarica*:* We denote this instance of StHorn as StHorn_E. In contrast to Spacer, Eldarica is based on Predicate Abstraction, Counterexample-Guided Abstraction Refinement (CEGAR) and Interpolation. In addition, Eldarica's API does not enable to load an interpretation for a predicate. Instead, it supports an incremental usage where solving can be invoked with a substitution map such that predicates are completely substituted with a given formula.

For using Eldarica in StHorn, we implemented a satisfiability-preserving transformation for CHCs. Let Π be the set of CHCs. The transformation uses additional predicates that are added in the following way. First, we add the set $\mathcal{P}^{\Pi}_{g} := \{p_g \mid p \in \mathcal{P}^{\Pi}\}$ that consists of a *ghost* predicate for every predicate in Π. Then, we add the set $\mathcal{P}^{\Pi}_{en} := \{p_\pi \mid \pi \in \Pi\}$ that consists of an *enable* predicate for every clause in Π. While ghost predicates have the same arity as their original counterparts, enable predicates have 0-arity (i.e., they are uninterpreted Boolean constants). The new set of uninterpreted predicates is $\mathcal{P}^{\Pi} \cup \mathcal{P}^{\Pi}_{g} \cup \mathcal{P}^{\Pi}_{en}$.

Next, the clauses are modified such that the enable predicate p_π is added (as a conjunct) to the body of every clause π. Then, if the body of a clause contains a p-formula $p(t_1, \ldots, t_n)$, where $p \in \mathcal{P}^{\Pi}$, the p_g-formula $p_g(t_1, \ldots, t_n)$ is added (as a conjunct) to the body as well. In this way, StHorn can use Eldarica's incremental API by supplying every call to the solver with a substitution map that substitutes every ghost predicate with its current rule-satisfying interpretation, and using the enable predicates to control what subset of clauses is being considered (in a similar manner to enable literals in SAT).

Remark 4. Importantly, while this transformation is satisfiability-preserving and allows StHorn to use any CHC-solver (even one that is not incremental), it is more limiting than what the Z3 API is allowing. The main reason is that using this method can only result in strengthening of the rule-satisfying interpretation, since the given substitutions are not modified by the solver. Both Spacer and Eldarica employ various optimizations that can help convergence. The above transformation may interfere with such optimizations. As an example, by employing "global guidance" [29], Spacer can generalize a set of lemmas that are already present in an interpretation of a predicate (during its execution). If we would have used the above transformation with Spacer, we would have most likely interfere with this optimization.

6.2 Experimental Evaluation

In this section, we present our experimental results. We used the CHC-COMP'22 benchmarks [13], and compared StHorn against Spacer and Eldarica. The comparison is done with respect to the corresponding instance. Namely, StHorn_S against Spacer and StHorn_E against Eldarica. For the comparison we used two categories: (1) linear clauses over the theory of Linear Integer Arithmetic (LIA), and (2) non-linear clauses over the theory of LIA. Overall, there are 499 CHC instances for the linear CHCs, and 456 non-linear CHCs instances. All experiments were executed on a workstation with AMD EPYC 74F3, a 24-Core CPU. Every instance was given 900 s and 8 GB of memory.

Table 1. Comparison of StHorn and Spacer

Benchmarks	Tool	Total		SAT		UNSAT		Hard	
		Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
Linear CHCs	Spacer	320 (7)	76.5	234	67.2	86	101.2	43	468.8
	StHorn$_S$	**322 (9)**	**67.4**	234	**63.4**	**88**	**78.2**	**45**	**411.2**
	portfolio	329	53.5	239	48.7	90	66.5	52	326.9
Non-Linear CHCs	Spacer	386 (2)	60.4	276	**48.8**	110	88.1	68	265.4
	StHorn$_S$	**406 (22)**	**48.9**	**286**	56.7	**120**	**30.2**	**88**	**198.4**
	portfolio	408	23.8	287	30.6	121	7.7	90	100.3

In the case of Z3, to increase the reliability of the evaluation and demonstrate that the results were not determined by random decisions made by Z3, all experiments were executed with three different random seeds (a Z3 parameter), and the results presented are an average of these runs.[6]

Table 1 and Table 2 summarize the experiments comparing StHorn with Spacer and Eldarica, respectively. The tables present both the total number of solved instances and the average run-time, as well as a distinction between satisfiable and unsatisfiable instances. The reported average runtimes only consider the instances that were solved by at least one of the tools (if both tools report "unknown", the instance is not counted). The numbers in brackets represent uniquely solved instances. In addition, both tables present results for *hard* instances, which are instances where at least one of the tools required at least 60 s to solve. Lastly, the tables also present the results of a portfolio solver. Namely, a solver that runs both variants simultaneously (StHorn$_S$ and Spacer for Table 1; StHorn$_E$ and Eldarica for Table 2) and halts when one of them terminates with a definitive result. In the following we analyze the results of both tables, divided by linear and non-linear CHCs instances.

StHorn$_S$ vs Spacer

Linear CHCs: In this category, StHorn$_S$ solves two more instances than Spacer and also performs better w.r.t. runtime (though the difference is not big). The set of instances they solve are also different as StHorn$_S$ solves 9 instances not solved by Spacer, while Spacer solves 7 instances not solved by StHorn$_S$.

Non-Linear CHCs: On these instances, StHorn$_S$ solves 20 more instances than Spacer. As can be seen from the table, the average runtime is in favor of StHorn$_S$. When further analyzing the results we discover that if one considers only unsatisfiable instances, not only StHorn$_S$ solves more instances, it also performs almost 3 times faster.

Portfolio: We also present the results for a portfolio solver that runs both StHorn$_S$ and Spacer simultaneously. From these results we see that the portfo-

[6] We were not able to find such a parameter for Eldarica.

Table 2. Comparison of StHorn and Eldarica

Benchmarks	Tool	Total		SAT		UNSAT		Hard	
		Solved	Time [s]	Solved	Time [s]	Solved	Time [s]	Solved	Time [s]
Linear CHCs	Eldarica	226 (18)	140.2	**156**	98.9	70	220	40	532.7
	StHorn$_E$	**231 (23)**	**108.7**	151	**97.5**	**80**	**130.2**	**45**	**403.3**
	portfolio	249	65.6	164	50	85	95.6	63	239.5
Non-Linear CHCs	Eldarica	**325 (40)**	**74.5**	**198**	**81.5**	**127**	**63.2**	**134**	**156.4**
	StHorn$_E$	302 (17)	168.3	194	129.1	108	231.5	111	366.7
	portfolio	342	27.3	211	19.1	131	40.5	151	53

lio solver shows a great improvement in runtime over each of the solvers alone, in both categories. This shows that StHorn$_S$ can complement Spacer.

StHorn$_E$ vs Eldarica

Linear CHCs: StHorn$_E$ performs better than Eldarica on the set of linear CHCs as it solves more instances and performs better w.r.t. runtime. In addition, the set of instances solved by each tool is different: StHorn$_E$ solves 23 instances not solved by Eldarica, while Eldarica solves 18 instances not solved by StHorn$_E$. Analyzing the instances based on their satisfiability shows that the biggest improvement is achieved on unsatisfiable instances (1.7 times faster).

Non-Linear CHCs: On these instances, however, Eldarica performs better than StHorn$_E$, on both number of solved instances and average runtime. A more detailed analysis of the results reveal that for StHorn$_E$, the time spent in the `Amend` procedure is significant. This has a few reasons. First, the interpretations returned by Eldarica are not necessarily a set of conjuncts, which limits its StHorn$_E$'s ability to retain parts of the satisfying interpretations returned when analyzing a subset of clauses. Second, since Eldarica does not have an API that allows "pre-loading" a rule-satisfying interpretation for a predicate, we used a satisfiability-preserving transformation. However, this transformation limits StHorn$_E$ (see Remark 4) such that it can only "strengthen" the given rule-satisfying interpretation when invoking `Solve`. Lastly, since StHorn$_E$ makes many calls to Eldarica through JNI, this imposes an overhead.

Portfolio: Despite all of the above, when considering a portfolio solver that invokes both StHorn$_E$ and Eldarica, performance improve quite significantly both in the number of solved instances and runtime. This again shows that StHorn$_E$ can complement Eldarica and improve its performance.

Summary. Overall, StHorn solved more instances and had a faster runtime than Spacer and Eldarica. One exception is the Non-linear category, where Eldarica outperforms StHorn. StHorn, however, demonstrated substantial improvements in the portfolio solver, both in the latter category and the rest, indicating that it complements both tools by allowing them to solve new instances more efficiently. In addition, our evaluation indicates a greater improvement for UNSAT

instances, but also a promising improvement for SAT instances. It is therefore evident that StHorn can improve upon the state-of-the-art in CHC solving.

7 Related Work

There is a large body of work on solving the CHC-SAT problem, with a plethora of algorithms and tools that are based on different methods such as IC3/PDR, interpolation, Counterexample-Guided Abstraction Refinement (CEGAR), Predicate Abstraction, and Machine Learning [7,8,12,14,16,23,24,28,34]. The technique presented in this paper, StHorn, is orthogonal to these algorithms as it uses a CHC-solver as a "black-box".

CHCs gained popularity in recent years since many program, and recently hardware, verification problems can be reduced to the satisfiability of CHCs [7,17,20,26,33]. Many program verification algorithms work by analyzing different paths in the program separately, when trying to establish the correctness of the whole program [10,21,22,31]. In this sense, StHorn draws its intuition from path-sensitive verification algorithms. However, most program verification algorithms that operate on paths consider bounded execution paths in the control flow graph, while StHorn considers complete paths in the graph, that may include loops. Intuitively, this is similar to analyzing complete fragments of a program that include loops, without unrolling them explicitly. The closest work to ours in this regard is [6] where complete fragments of a program (i.e., "path programs") are considered. The usage, however, is quite different as they use "path invariants" to eliminate spurious counterexamples in the context of CEGAR, whereas we construct satisfying interpretations for CHC sets incrementally based on interpretations of satisfiable subsets.

Hypergraphs have been suggested before in [2] for solving propositional Horn formulas, in which the uninterpreted predicate symbols are Boolean. That is, they can be assigned either $\top$ or $\bot$. Given a propositional Horn formula, they show how to maintain on-line information about its satisfiability during the insertion of new clauses. Clearly, this is a different problem.

Lastly, StHorn uses a structure-guided heuristic for selecting the subsets to be solved and tries to re-use information when analyzing different subsets. We are unaware of a similar heuristic for prioritizing clauses during the search for a satisfying interpretation.

8 Conclusion

In this work, we present StHorn, a technique for deciding the satisfiability of a set Π of CHCs. StHorn handles monotonically larger subsets of Π, which are selected based on its structure. The technique exploits a satisfying interpretation obtained for one subset as a basis for solving subsequent subsets. We use a CHC-solver as a "black-box". Our evaluation shows that StHorn, when added on top of Spacer, improves upon state-of-the-art. Moreover, it complements both Spacer and Eldarica, allowing them to solve new instances more efficiently.

Future research plans include: (i) designing domain-oriented selection strategies; (ii) enhancing current (syntactic) strategies with semantic hints; and (iii) integrating the technique natively into a CHC-solver, reducing the overhead imposed by its API and further improving performance.

References

1. Ausiello, G., Franciosa, P.G., Frigioni, D.: Directed hypergraphs: problems, algorithmic results, and a novel decremental approach. In: ICTCS 2001. LNCS, vol. 2202, pp. 312–328. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45446-2_20
2. Ausiello, G., Italiano, G.F.: On-line algorithms for polynomially solvable satisfiability problems. J. Log. Program. **10**(1), 69–90 (1991). https://doi.org/10.1016/0743-1066(91)90006-B
3. Ausiello, G., Italiano, G.F., Nanni, U.: Optimal traversal of directed hypergraphs. Technical report, TR-92-073 (1992)
4. Ausiello, G., Italiano, G.F., Nanni, U.: Hypergraph traversal revisited: cost measures and dynamic algorithms. In: Brim, L., Gruska, J., Zlatuška, J. (eds.) MFCS 1998. LNCS, vol. 1450, pp. 1–16. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0055754
5. Beyene, T.A., Popeea, C., Rybalchenko, A.: Efficient CTL verification via horn constraints solving. In: Gallagher, J.P., Rümmer, P. (eds.) Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3 April 2016. EPTCS, vol. 219, pp. 1–14 (2016). https://doi.org/10.4204/EPTCS.219.1
6. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, 10–13 June 2007, pp. 300–309. ACM (2007). https://doi.org/10.1145/1250734.1250769
7. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2
8. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38856-9_8
9. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
10. Das, M., Lerner, S., Seigle, M.: ESP: path-sensitive program verification in polynomial time. In: Knoop, J., Hendren, L.J. (eds.) Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, 17–19 June 2002, pp. 57–68. ACM (2002). https://doi.org/10.1145/512529.512538
11. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained horn clauses for program verification. Theory Pract. Logic Program. **22**(6), 974–1042 (2022). https://doi.org/10.1017/S1471068421000211

12. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: VeriMAP: a tool for verifying programs through transformations. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 568–574. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_47
13. De Angelis, E., Govind, V.K.H.: CHC-COMP 2022: competition report. In: Hamilton, G.W., Kahsai, T., Proietti, M. (eds.) Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022. EPTCS, vol. 373, pp. 44–62 (2022). https://doi.org/10.4204/EPTCS.373.5
14. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: Stewart, D., Weissenbacher, G. (eds.) 2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, 2–6 October 2017, pp. 100–107. IEEE (2017). https://doi.org/10.23919/FMCAD.2017.8102247
15. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): a software verifier based on horn clauses. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-28756-5_46
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Vitek, J., Lin, H., Tip, F. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, Beijing, China, 11–16 June 2012, pp. 405–416. ACM (2012). https://doi.org/10.1145/2254064.2254112
17. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: a constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_32
18. Gurfinkel, A.: Program verification with constrained horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) CAV 2022. LNCS, vol. 13371, pp. 19–29. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_2
19. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015, Part I. LNCS, vol. 9206, pp. 343–361. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_20
20. Gurfinkel, A., Shoham, S., Meshman, Y.: SMT-based verification of parameterized systems. In: Zimmermann, T., Cleland-Huang, J., Su, Z. (eds.) Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, 13–18 November 2016, pp. 338–348. ACM (2016). https://doi.org/10.1145/2950290.2950330
21. Harris, W.R., Sankaranarayanan, S., Ivancic, F., Gupta, A.: Program analysis via satisfiability modulo path programs. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, 17–23 January 2010, pp. 71–82. ACM (2010). https://doi.org/10.1145/1706299.1706309
22. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January 2004, pp. 232–244. ACM (2004). https://doi.org/10.1145/964001.964021

23. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_13
24. Hojjat, H., Rümmer, P.: The ELDARICA horn solver. In: Bjørner, N., Gurfinkel, A. (eds.) 2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, 30 October–2 November 2018, pp. 1–7. IEEE (2018). https://doi.org/10.23919/FMCAD.2018.8603013
25. Hojjat, H., Rümmer, P., McClurg, J., Cerný, P., Foster, N.: Optimizing horn solvers for network repair. In: Piskac, R., Talupur, M. (eds.) 2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, 3–6 October 2016, pp. 73–80. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886663
26. Govind, H.V.K., Fedyukovich, G., Gurfinkel, A.: Word level property directed reachability. In: IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, 2–5 November 2020, pp. 107:1–107:9. IEEE (2020). https://doi.org/10.1145/3400302.3415708
27. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: JayHorn: a framework for verifying java programs. In: Chaudhuri, S., Farzan, A. (eds.) CAV 2016. LNCS, vol. 9779, pp. 352–358. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-41528-4_19
28. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods Syst. Des. **48**(3), 175–205 (2016). https://doi.org/10.1007/s10703-016-0249-4
29. Vediramana Krishnan, H.G., Chen, Y.T., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 101–125. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_7
30. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for rust programs. ACM Trans. Program. Lang. Syst. **43**(4), 15:1–15:54 (2021). https://doi.org/10.1145/3462205
31. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006). https://doi.org/10.1007/11817963_14
32. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
33. Zhang, H., Gupta, A., Malik, S.: Syntax-guided synthesis for lemma generation in hardware model checking. In: Henglein, F., Shoham, S., Vizel, Y. (eds.) VMCAI 2021. LNCS, vol. 12597, pp. 325–349. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-67067-2_15
34. Zhu, H., Magill, S., Jagannathan, S.: A data-driven CHC solver. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, 18–22 June 2018, pp. 707–721. ACM (2018). https://doi.org/10.1145/3192366.3192416