ISCAS-SKLCS-15-19 **Dec., 2015**

中国科学院软件研究所
计算机科学国家重点实验室
技术报告

Trace Abstraction Refinement for Solving Horn Clauses

by

Weifeng Wang, Li Jiao

State key Laboratory of Computer Science
Institute of Software
Chinese Academy of Sciences
Beijing 100190. China

Trace Abstraction Refinement for Solving Horn Clauses

Weifeng Wang[1,2] and Li Jiao[1]

[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China
{wangwf,ljiao}@ios.ac.cn
[2] University of Chinese Academy of Sciences, Beijing 100049, China

Abstract. Horn clauses can be used in many areas such as logic programming, artificial intelligence and formal methods. Horn clause solving is closely related to program verification. On the one hand, program verification tasks can be translated to Horn clause solving problems. On the other hand, Horn clause solving tasks can be accomplished using some of the program verification techniques. As a result, Horn clauses can be used as an intermediate language in program verification, decoupling the verification algorithms from the details of the specific programming languages. In this paper, we propose a novel method for solving Horn clauses, which is inspired by a program verification method called trace abstraction refinement. In our method, solvability of Horn clauses is verified by alternatively analyzing its unfoldings and constructing and manipulating tree automata. Since Horn clauses can serve as an intermediate language for program verification, our method generalizes the original trace abstraction refinement algorithm, making it easier to be used for various program verification tasks. We illustrate some of the existing works on how to reduce safety verification tasks of multi-threaded programs and programs with procedures to Horn clause solving problems. Preliminary experimental results are reported.

1 Introduction

Nowadays the counterexample guided abstraction refinement (CEGAR) [1] idea has been popular for program verification. A CEGAR algorithm performs abstract model construction and model checking alternatively, trying to avoid checking the original, detailed model, which is generally large in size. Lazy abstraction [2] improves CEGAR by performing these two steps simultaneously, i.e., constructing and refining the abstract models on-the-fly during the model checking phase. Interpolation techniques [3] can be used to generate predicates for the refinement.

While many of the works focus on state-based abstractions, Heizmann et al.[4] proposed a trace-based abstraction refinement scheme. For the trace abstraction refinement method, the behavior of a program is represented by the set of traces it can execute. Regular languages, represented by finite automata,

are used to abstract (over-approximate) the behavior of programs. The abstractions are incrementally refined by constructing new regular languages. which only contain infeasible traces, and subtracting them from the previous abstract models.

Horn clauses are a subclass of logic formulas, which are extensively used in various areas of computer science such as logic programming, artificial intelligence, formal methods, etc. Horn clause solving (or equivalently Constraint Logic Program (CLP) [5]-proving) is closely related to program verification. On one hand, program verification tasks can be encoded into Horn clause-solving problems [6]. On the other hand, the problem of Horn clause solving could be tackled using various CEGAR-based program verification techniques [6–9]. As a result, Horn clauses can serve as an intermediate representation of software verification problems.

The idea of using Horn clauses as intermediate representation of software verification problems has been proposed by Grebenshchikov et al. [6]. As a result, developing a program verifier consists of two steps: i) engineering a backend Horn clause solver, and ii) constructing a frontend which automatically encodes software verification problems into Horn clause solving problems. The benefit is three-fold. Firstly, if one wants to build a verifier for a new programming language, he/she does not have to build it from the ground up. He/she just needs to construct a frontend for the application domain, while existing backend Horn clause solvers can be reused. Secondly, if one wants to try a new verification algorithm, he/she does not have to know the specific application domain. He/she just has to build the backend Horn clause solver, while existing frontends for various application domains can be reused. Thirdly, as stated in [10], by using different frontend encoding schemes, the verification procedure could be guided or constrained as needed, which provides flexibility. This last point can also be shown in [6, 11], where compositional reasoning rules such as Owicki-Gries rules [12] and rely-guarantee rules [13] are conveniently embedded in the Horn clause encoding, facilitating compositional verification of multi-threaded programs, and modular reasoning rules are used for encoding programs, facilitating modular verification of programs with procedure calls.

In this paper, we follow the idea of adapting CEGAR-based methods for Horn clause solving. We propose a trace abstraction refinement based method for Horn clause solving. We prove that a set of Horn clauses is solvable iff all of its *ground unfoldings* are solvable. The latter can be automatically checked in an incremental approach. We use tree automata to collect the ground unfoldings of the set of Horn clauses. Initially a tree automaton is constructed, which captures all ground unfoldings of the set of Horn clauses. In each iteration, a ground unfolding is picked and checked whether it is solvable. If it is not solvable then the procedure terminates with the conclusion that the set of Horn clauses is not solvable, otherwise a tree automaton is constructed, which collects this ground unfolding and possibly other solvable ground unfoldings. This tree automaton is used in the next steps to prevent the set of ground unfoldings represented by it from being picked again. The above iteration continues until there is no ground

unfolding to pick, which means that all the ground unfoldings are proved to be solvable. In this case we will get the conclusion that the set of Horn clauses is solvable.

Tree interpolation techniques are used during the construction of tree automata. Tree automata have the nice property of closure under union, intersection and complement, which makes it possible to solve Horn clauses by manipulating tree automata.

Our work results in a generalized version of the trace abstraction refinement algorithm, which can be easily used for performing various program verification tasks when combined with corresponding frontend encoders.

We have implemented a prototype and evaluated the performance on some Horn clause benchmarks. The result shows that the trace abstraction refinement method needs more iterations to terminate than predicate abstraction based CEGAR, while each iteration takes less time. Although the overall performance is not so good as predicate abstraction based CEGAR algorithm, we point out possible improvements.

Related works. Horn clause has been studied for a long time. A detailed survey of Horn clauses and Constraint Logic Programming could be found in [5].

Trace abstraction refinement was initially proposed by Heizmann et al. [4, 14]. The authors used finite automata as abstractions (over-approximations) of program behavior, which is limited to simple programs. Several works have been devoted on extending this method for verifying more expressive classes of programs. Heizmann et al. [15] extended this scheme by using nested word automata instead of finite automata to verify programs with recursive procedures. Cassez et al. [16] proposed a summary-based modular trace refinement method for verifying programs with procedures. Farzan et al. [17] used counter automata instead to extend this scheme for the verification of parameterized systems. We replace finite automata with tree automata in trace abstraction refinement for Horn clause solving. Verification of multi-threaded programs and programs with procedures can be reduced to Horn clause solving problems [6]. Since the resulting Horn clause encodings are typically non-linear, while the original trace abstraction refinement method can only be applied on linear Horn clauses, a consequence of our work is a generalization of the trace abstraction refinement scheme to handle the verification tasks of richer classes of programs. Moreover, we can see that certain efforts have been devoted to adapt trace abstraction refinement for verifying programs with procedures [15, 16], while as a result of our work, the task of modular verification of programs with procedures and compositional verification of multi-threaded programs are just a matter of combining with corresponding frontend encoding schemes.

As is mentioned in [18], current SAT (satisfiability) based verification technique is based on two major approaches: interpolation based verification and Property Directed Reachability(PDR)(a.k.a. IC3) [19]. These two techniques have been adapted for Horn clause solving. Hoder et al. [8] extended PDR for solving Horn clauses. The contribution is two-fold: they proposed a new method for Horn clause solving, and generalized PDR for more expressive systems such

as timed pushdown systems. Grebenshchikov et al. [6] suggested to solve Horn clauses using predicate abstraction, where tree interpolation techniques are used to generate predicates for the refinement. Rümmer et al. [7] proposed to use disjunctive interpolation instead of tree interpolation for Horn clause solving, which can handle multiple traces simultaneously and reduce the number of iterations. McMillan et al.[9] proposed to solve Horn clauses using interpolation only – without predicate abstraction. We extend one of the interpolation-based program verification techniques called *trace abstraction refinement* for Horn clause solving. Similar to [8], our work can be seen as both a generalization of an existing program verification technique and a new method of Horn clause solving.

Kafle et al. [20] have also proposed the idea to relate a set of Horn clauses to a tree automaton. Both of our works use tree automata operations to exclude infeasible traces. In their work, the tree automaton constructed in each iteration accepts only one infeasible trace (although the possibility of constructing a tree automaton which accepts more than one infeasible traces is mentioned in their paper). The purpose of the tree automata difference operation is mainly to obtain a "reshaped" set of Horn clauses (a new set of Horn clauses is generated according to the tree automaton obtained by the tree automata difference operation in each iteration). The "reshape" steps interact with the abstract interpretation steps, resulting in improved performance over the pure abstract interpretation method. In our work, the tree automaton constructed in each iteration accepts more than one (possibly infinitely many) traces. The tree automaton is constructed according to the tree interpolant generated by interpolating SMT solver. In fact, the automaton constructed in each iteration is just used to collect the traces that have been proved infeasible by the SMT solver. The difference automaton in each iteration represents all traces that still need to be checked. Our algorithm is more close to the original idea in [4]. Moreover, the complexity issue of tree automata determinisation and completion operations is addressed in [20] by refering to optimized algorithms in [21].

Organization of the paper. The rest of this paper is organized as follows. We give a motivating example in Section 2 to show our idea. In Section 3 basic concepts of Horn clauses and tree automata are described. Solvability of Horn clauses is discussed in Section 4, which provides the basis for our Horn clause solving algorithm. In Section 5 we present a detailed description on how to use trace abstraction refinement for Horn clause solving, and prove the correctness of our method. In Section 6, we explain how verification tasks of multi-threaded programs and programs with procedures can be encoded to Horn clause solving problems, showing that our method can be seen as an extension of the original trace abstraction refinement scheme. We present the experiment results in Section 7. Finally, a conclusion is given in Section 8.

2 A motivating example

As a motivating example, let us consider a set of linear Horn clauses obtained from program verification tasks. By "linear" we mean that there is at most one

uninterpreted predicate in the body of each Horn clause. Given an arbitrary program with the set of variables $\mathbf{x}$, the initial condition ϕ_{init} and the transition relation ϕ_{trans}, we want to prove that the unsafe condition ϕ_{unsafe} is never reached. We could encode the program as a set of Horn clauses:

$(h1)\ Inv(\mathbf{x}) \leftarrow \phi_{init}(\mathbf{x})$
$(h2)\ Inv(\mathbf{x}') \leftarrow Inv(\mathbf{x}) \wedge \phi_{trans}(\mathbf{x}, \mathbf{x}')$
$(h3)\ \mathit{false} \leftarrow Inv(\mathbf{x}) \wedge \phi_{unsafe}(\mathbf{x})$

where Inv is an uninterpreted predicate, and $\mathbf{x}'$ is a tuple of primed variables corresponding to those in $\mathbf{x}$ and representing variables in the next state. We can see that the program is safe iff the above set of Horn clauses is consistent, i.e., there is an interpretation ϕ_{Inv} of Inv such that all the above Horn clauses are satisfied. Here we could regard "*false*" as a special uninterpreted predicate which is expected to be interpreted as *false*.

Intuitively, this set of Horn clauses corresponds to a finite automaton whose alphabet is $\{h1, h2, h3\}$, and whose set of states is $\{init, Inv, \mathit{false}\}$. The corresponding finite automaton is shown in Figure 1. The set of final states is $\{\mathit{false}\}$. Each accepted trace of this finite automaton corresponds to a derivation sequence of the set of Horn clauses. For example, the trace $h1h2h2h3$ corresponds to the derivation sequence:

$\phi_{init}(\mathbf{x}_0) \rightarrow Inv(\mathbf{x}_0)$
$Inv(\mathbf{x}_0) \wedge \phi_{trans}(\mathbf{x}_0, \mathbf{x}_1) \rightarrow Inv(\mathbf{x}_1)$
$Inv(\mathbf{x}_1) \wedge \phi_{trans}(\mathbf{x}_1, \mathbf{x}_2) \rightarrow Inv(\mathbf{x}_2)$
$Inv(\mathbf{x}_2) \wedge \phi_{unsafe}(\mathbf{x}_2) \rightarrow \mathit{false}$

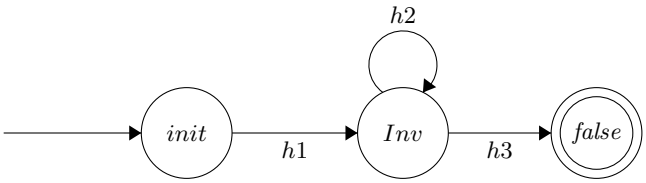

Fig. 1: The finite automaton for the set of Horn clauses $\{h1, h2, h3\}$.

For linear Horn clauses, we could use finite automata to represent its derivations. In [4], the authors proposed a trace abstraction refinement scheme, which represents traces of programs as regular patterns, and verifies a program by constructing and manipulating regular patterns of traces. Given the correspondence between programs and linear Horn clauses described above, the trace abstraction refinement method could be naturally adapted to verify the solvability of linear Horn clauses.

Note that in the correspondence between a set of Horn clauses and a finite automaton, each uninterpreted predicate (including "*false*") corresponds to a state and each Horn clause corresponds to a transition. Since there is at most one predecessor state in each transition, it is important that each Horn clause contains at most one appearance of an uninterpreted predicate. For non-linear Horn clauses, there might be more than one appearance of uninterpreted predicates in the body, which makes it impossible to correspond to a finite automaton.

Now consider the following set of Horn clauses

$(h1)$ $p(x, y) \leftarrow x = y$
$(h2)$ $p(x, z) \leftarrow p(x, y) \wedge z = y + 1$
$(h3)$ $q(x, z) \leftarrow p(x, y) \wedge p(y, z)$
$(h4)$ $\mathit{false} \leftarrow q(x, y) \wedge x > y$

Following the intuition in the linear case, the alphabet is $\{h1, h2, h3, h4\}$. However, the clause $h3$ does not correspond to any transition in a finite automaton, because the uninterpreted predicate p appears twice in the body, which corresponds to *two* states as the predecessors. Moreover, each derivation corresponds to a *tree* rather than a sequence (see Figure 2). In order to cope with this situation, we use *tree automata* (specifically, automata on ranked trees), which are an extension of finite automata, to represent non-linear Horn clauses. In a tree automaton, each transition can have more than one predecessor states, which is sufficient to represent a non-linear Horn clause.

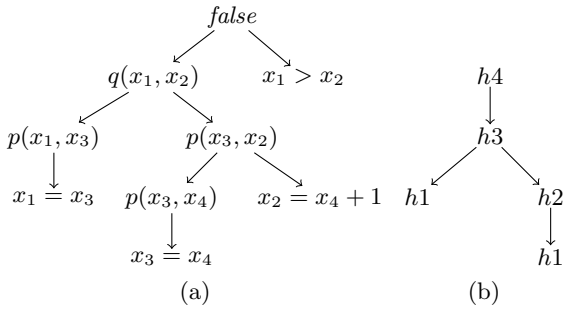

Fig. 2: (2a) a derivation to *false*, and (2b) the corresponding trace, which is a tree rather than a sequence.

In order to verify the solvability of non-linear Horn clauses, we should extend the trace abstraction refinement method by using tree automata rather than finite automata to represent sets of traces. In the following sections we will give a detailed description of the extension.

3 Preliminaries

3.1 Constraint language

We use first order logic with a set $\mathcal{F}$ of interpreted functions and a set $\mathcal{P}$ of interpreted predicates as the underlying constraint language. The symbols in $\mathcal{F}$ and $\mathcal{P}$ are interpreted by a structure (U, I), where U is a non-empty universe, and I assigns to each function in $\mathcal{F}$ a function over U, and to each predicate in $\mathcal{P}$ a relation over U. Moreover, the existence of the equation symbol "=" with conventional interpretation is assumed.

Given a formula ϕ with free variables $x_1, x_2, \ldots, x_n$, we define the *universal closure* $Cl_\forall(\phi)$ of ϕ as $\forall x_1, \ldots, x_n.\phi$. We denote the set of free variables for a formula ϕ as $fv(\phi)$.

For two contradicting formulas ϕ_A and ϕ_B (i.e., $\phi_A \wedge \phi_B$ is not satisfiable), an interpolant is a formula ϕ_I such that ϕ_A implies ϕ_I, and that ϕ_I contradicts ϕ_B, and $fv(\phi_I) \subseteq fv(\phi_A) \cap fv(\phi_B)$.

Our results are independent of specific theories (e.g., linear integer arithmetic, linear real arithmetic) of the constraint language, as long as there is an algorithm to compute interpolants.

3.2 Horn clauses

Following [6], we make the definition of Horn clauses slightly different from the standard notion. In our definition, constraints appearing in a Horn clause can have both disjunctions and conjunctions.

Definition 1 (Horn clause). *Let $\mathcal{R}$ be a set of uninterpreted predicates disjoint from $\mathcal{F}$ and $\mathcal{P}$, and $\mathcal{X}$ a set of variables, a Horn clause is a logic formula of the form $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$, where*

- *C is a formula in the constraint language over $\mathcal{F}$, $\mathcal{P}$ and $\mathcal{X}$, and*
- *each B_i is an application $p_i(x_1, \ldots, x_k)$ of an uninterpreted predicate $p_i \in \mathcal{R}$ over the variables $x_1, \ldots, x_k \in \mathcal{X}$, and*
- *H is either an application of $p_h \in \mathcal{R}$ to the variables in $\mathcal{X}$, or the formula* false*.*

It is worth noting that, this definition of Horn clauses is equivalent to Constraint Logic Programs (CLP) [5]. Moreover, all of the concepts and results in Section 4 have their correspondences in CLP.

In the sequel, we might write a Horn clause in a reverse order, i.e., $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ for convenience. We abbreviate a tuple of free variables $(x_1, x_2, \ldots, x_k)$ as $\mathbf{x}$ for an uninterpreted predicate with k parameters. Then B_i could be rewritten as $p_i(\mathbf{x}_i)$, and H as $p_h(\mathbf{x}_h)$, or $\mathit{false}(\mathbf{x}_h)$. Here, for convenience, we regard *false* as a special uninterpreted predicate, whose tuple of parameters $\mathbf{x}_h$ contains 0 variables. The constraint C contains free variables in $\mathbf{x_1}, \ldots, \mathbf{x}_n$ and $\mathbf{x}_h$ and other variables in $\mathcal{X}$. We denote the tuple of free variables that only occur in C as $\mathbf{x}_0$. Consequently, the constraint C can also be written

as $C[\mathbf{x}_0, \ldots, \mathbf{x}_n, \mathbf{x}_h]$ to address the fact that the free variables in $\mathbf{x}_0, \ldots, \mathbf{x}_n, \mathbf{x}_h$ appear in C.

The definition of Horn clauses described above is restrictive. However, since we assume that the constraint language contains the equation symbol "=", it can be easily seen that other Horn clauses in the constraint language can be converted to equivalent restricted Horn clauses.

3.3 Tree automata

We give a brief introduction to tree automata, which are a generalization of finite automata. There are several different ways to define tree automata, among which we choose the nondeterministic bottom-up tree automata. A detailed description of tree automata and tree languages can be found in [22].

We denote by Σ a finite set of *function symbols*, where each function symbol $f \in \Sigma$ has an *arity* $n \in \mathbb{N}$. Let V be a set of variables. The set $T(\Sigma, V)$ of *terms* (a.k.a. *trees*) is defined inductively by:

- $v \in T(\Sigma, V)$ for each $v \in V$, and
- $f(t_1, \ldots, t_n) \in T(\Sigma, V)$ for an n-ary function $f \in \Sigma$, and the terms $t_1, \ldots, t_n \in T(\Sigma, V)$.

$T(\Sigma, \emptyset)$ is the set of *ground terms* over Σ.

Definition 2 (Tree automata). *A* tree automaton $\mathcal{A}$ *is a tuple* (Q, Σ, F, Δ)*, where* Q *is the set of states,* Σ *is the set of function symbols,* $F \subseteq Q$ *is the set of final states, and* Δ *is a finite set of transitions of the form* $f(q_1, \ldots, q_n) \to q$*, where* $f \in \Sigma$ *is an* n*-ary function symbol, and* $q, q_1, \ldots, q_n \in Q$*.*

A tree $t = f(t_1, \ldots, t_n) \in T(\Sigma, \emptyset)$ *is recognized at* $q \in Q$ *in* $\mathcal{A}$ *iff there is a transition* $f(q_1, \ldots, q_n) \to q \in \Delta$ *such that* t_i *is recognized at* q_i *in* $\mathcal{A}$*.* t *is accepted by* $\mathcal{A}$ *iff it is recognized at some final state* $q_f \in F$ *in* $\mathcal{A}$*.*

Similar to finite automata, tree automata are also closed under language-theoretic union, intersection and complement.

4 Solvability

Let $\mathcal{HC}$ be a set of Horn clauses, and Π a map which assigns to each uninterpreted predicate a formula in the constraint language having the same number of variables as parameters. Π could be extended to map a Horn clause to a formula in the constraint language as follows:
$\Pi(C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \to p_h(\mathbf{x}_h)) = C \wedge \Pi(p_1)(\mathbf{x}_1) \wedge \cdots \wedge \Pi(p_n)(\mathbf{x}_n) \to \Pi(p_h)(\mathbf{x}_h)$
$\Pi(C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \to \mathit{false}) = C \wedge \Pi(p_1)(\mathbf{x}_1) \wedge \cdots \wedge \Pi(p_n)(\mathbf{x}_n) \to \mathit{false}$

Definition 3 (Solvability [7]). *Given a structure* (U, I)*, A set* $\mathcal{HC}$ *of Horn clauses is said to be*

- syntactically solvable, *if there is a map* Π *such that* $(U, I) \models Cl_\forall(\Pi(h))$ *for each* $h \in \mathcal{HC}$*; and*
- semantically solvable, *if there is an interpretation* I_u *that assigns to each uninterpreted predicate a relation on* U*, such that for each Horn clause* $h \in \mathcal{HC}$*,* $(U, I \oplus I_u) \models Cl_\forall(h)$.

Here $I \oplus I_u$ is the disjoint union of I and I_u. In the sequel, we will always assume a fixed structure (U, I).

For convenience, Π could be extended to assign to the special uninterpreted predicate *false* the formula *false*.

Semantic solvability is related to the consistency of the set of Horn clauses (i.e., whether there is a contradiction), while syntactic solvability concerns whether there is a solution for the set of Horn clauses that is representable in the underlying constraint language.

We can easily see that, if a set of Horn clauses is syntactically solvable, then it is semantically solvable. However, the reverse is not necessarily true [7]. Next we investigate the sufficient and necessary conditions for the solvability of a set of Horn clauses.

For a set $\mathcal{HC}$ of Horn clauses, we define a *dependence* relation $\rightarrow_{\mathcal{HC}}$ on $(\mathcal{R} \cup \{false\})$: $p_1 \rightarrow_{\mathcal{HC}} p_2$ iff there is a Horn clause in $\mathcal{HC}$ containing p_1 in its head and p_2 in its body. We say that $\mathcal{HC}$ is *recursion-free* if $\rightarrow_{\mathcal{HC}}$ is acyclic, and *tree-structured* if $\rightarrow_{\mathcal{HC}}$ forms a tree and p appears at most once in h for each uninterpreted predicate $p \in \mathcal{R}$ and each Horn clause $h \in \mathcal{HC}$.

A tree-structured set of Horn clauses $\mathcal{HC}'$ is an *unfolding* of a set of Horn clauses $\mathcal{HC}$ if there is a mapping $\Gamma : \mathcal{R}_{\mathcal{HC}'} \rightarrow \mathcal{R}_{\mathcal{HC}}$ from the set of uninterpreted predicates of $\mathcal{HC}'$ to the set of uninterpreted predicates of $\mathcal{HC}$ such that

- for each Horn clause $C \wedge p'_1(\mathbf{x}_1) \wedge \cdots \wedge p'_n(\mathbf{x}_n) \rightarrow p'_h(\mathbf{x}_h) \in \mathcal{HC}'$, there is a Horn clause $C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow p_h(\mathbf{x}_h) \in \mathcal{HC}$ such that $\Gamma(p'_i) = p_i$ for $i \in \{1, \ldots, n\} \cup \{h\}$; and
- there is a unique Horn clause $C \wedge p'_1(\mathbf{x}_1) \wedge \cdots \wedge p'_n(\mathbf{x}_n) \rightarrow false$ in $\mathcal{HC}'$ such that $C \wedge \Gamma(p'_1)(\mathbf{x}_1) \wedge \cdots \wedge \Gamma(p'_n)(\mathbf{x}_n) \rightarrow false \in \mathcal{HC}$.

By the definition above we can see that Γ can be easily extended to map each Horn clause in $\mathcal{HC}'$ to a Horn clause in $\mathcal{HC}$.

Moreover, $\mathcal{HC}'$ is a *ground unfolding* if for each uninterpreted predicate $p' \in \mathcal{P}_{\mathcal{HC}'}$, there is a Horn clause $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H \in \mathcal{HC}'$, where H is $p'(\mathbf{x}_h)$ and $n \geq 0$. Intuitively, an unfolding $\mathcal{HC}'$ is a ground unfolding if each uninterpreted predicate appears in the head of some Horn clause in $\mathcal{HC}'$. Each ground unfolding corresponds to a derivation of *false*.

Example 1. For the set of Horn clauses $\mathcal{HC}_1$

$(h1)$ $p(x, y) \leftarrow x = y$
$(h2)$ $p(x, z) \leftarrow p(x, y) \wedge z = y + 1$
$(h3)$ $q(x, z) \leftarrow p(x, y) \wedge p(y, z)$
$(h4)$ $false \leftarrow q(x, y) \wedge x > y$

one of its unfoldings $\mathcal{HC}'_1$ is

(h'_1) $p_3(x, y) \leftarrow x = y$
(h'_2) $p_2(x, z) \leftarrow p_3(x, y) \wedge z = y + 1$
(h'_3) $p_1(x, y) \leftarrow x = y$
(h'_4) $q_1(x, z) \leftarrow p_1(x, y) \wedge p_2(y, z)$
(h'_5) $\mathit{false} \leftarrow q_1(x, y) \wedge x > y$

where for the mapping Γ, we have $\Gamma(p_1) = \Gamma(p_2) = \Gamma(p_3) = p$, and $\Gamma(q_1) = q$. We can see that this unfolding is in fact a ground unfolding.

Moreover, this unfolding corresponds to a derivation of *false*, where

- *false* is derived from $q(x_1, x_2) \wedge x_1 > x_2$ (corresponding to h'_5);
- $q(x_1, x_2)$ is derived from $p(x_1, x_3) \wedge p(x_3, x_2)$ (corresponding to h'_4);
- $p(x_1, x_3)$ is derived from $x_1 = x_3$ (corresponding to h'_3);
- $p(x_3, x_2)$ is derived from $p(x_3, x_4) \wedge x_2 = x_4 + 1$ (corresponding to h'_2);
- $p(x_3, x_4)$ is derived from $x_3 = x_4$ (corresponding to h'_1).

This derivation is illustrated in Figure 2a, where each node is derived by conjunctions of its children according to one of the Horn clauses.

In the following theorem, we establish the relation between the semantic solvability of a set of Horn clauses and that of its ground unfoldings.

Theorem 1. *A set of Horn clauses is semantically solvable, iff all of its ground unfoldings are semantically solvable.*

Theorem 1 is closely related to the soundness and completeness of top-down derivations in CLP [5]. However, here we develop it in a notation different from [5] for our special purpose.

In order to prove this theorem, we define a concept *reachable set* for uninterpreted predicates. For a set of Horn clauses $\mathcal{HC}$, the *reachable set* $\mathcal{I}_S$ assigns to each n-ary uninterpreted predicate (including *false*) an n-ary relation over the universe U. We define a special 0-ary tuple ϵ to handle the 0-ary uninterpreted predicates (including *false*). Basically, the relation $\mathcal{I}_S(p_i)$ for each p_i is the smallest relation satisfying

- *Rule* (i). For each tuple $\mathbf{t}_h$, each element of which is in U, if there is a Horn clause $C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow p_h(\mathbf{x}_h) \in \mathcal{HC}$, and a set of tuples $\mathbf{t}_i \in \mathcal{I}_S(p_i)$ for $i = 1, 2, \ldots, n$, such that $(\mathbf{t}_0, \mathbf{t}_1, \ldots, \mathbf{t}_n, \mathbf{t}_h) \in I(C)$, and $(\mathbf{t}_i) \in \mathcal{I}_S(p_i)$, then $\mathbf{t}_h \in \mathcal{I}_S(p_h)$.

Here $I(C)$ denotes the k-ary relation of C when interpreted on the structure (U, I), where k is the sum of numbers of variables in the tuples $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$.

$\mathcal{I}_S$ corresponds to the least fixedpoint semantics of Constraint Logic Programs [5], while Rule (i) corresponds to the one-step consequence operator. We call $\mathcal{I}_S$ the *reachable set* because it resembles to some extent the reachable state sets of programs, which can also be seen as a kind of least fixedpoint.

It is easy to prove that, if there is a semantic solution $\mathcal{I}$ for $\mathcal{HC}$, then it must be the case that $\mathcal{I}_S(p) \subseteq \mathcal{I}(p)$ for each uninterpreted predicate p. Moreover, for an unfolding $\mathcal{HC}'$ and its corresponding reachable set $\mathcal{I}'_S$, it is obvious that $\mathcal{I}'_S(p') \subseteq \mathcal{I}_S(\Gamma(p'))$ for each uninterpreted predicate p' in $\mathcal{HC}'$.

As stated previously, *false* can be seen as a special uninterpreted predicate. For an arbitrary semantic solution I_u of $\mathcal{HC}$, I_u can be naturally extended to assign to *false* a set of 0-tuples according to Rule (i) described above. By the definition of solvability, we know that it must be the case that $I_u(\mathit{false}) = \emptyset$, i.e., $I_u(\mathit{false}) \neq \{\epsilon\}$.

Lemma 1. *A set of Horn clauses is semantically solvable iff* $\mathcal{I}_S(\text{false}) = \emptyset$.

Proof. $\Rightarrow$. If there is a semantic solution I_u, then we could get $\mathcal{I}_S(\mathit{false}) \subseteq I_u(\mathit{false}) = \emptyset$.

$\Leftarrow$: if $\mathcal{I}_S(\mathit{false}) = \emptyset$, then according to Definition 3 $\mathcal{I}_S$ is a solution.

Intuitively, a set of Horn clauses is solvable iff *false* can not be "reached". Lemma 1 states the relation between the semantic solvability of a set of Horn clauses and the property of the corresponding reachable set. In the following lemma we establish the relation between the properties of the reachable set of a set of Horn clauses and the semantic solvability of its ground unfoldings.

Lemma 2. *There is a semantically unsolvable ground unfolding iff* $\mathcal{I}_S(\text{false}) \neq \emptyset$ *(*$\mathcal{I}_S(\text{false}) = \{\epsilon\}$*).*

Proof. $\Rightarrow$.For the unsolvable unfolding $\mathcal{HC}'$, we can construct a reachable set $\mathcal{I}'_S$. For each uninterpreted predicate p' appearing in $\mathcal{HC}'$, we have $\mathcal{I}'_S(p') \subseteq \mathcal{I}_S(\Gamma(p))$. Since $\mathcal{HC}'$ is unsolvable, by Lemma 1 we have $\emptyset \neq \mathcal{I}'_S(\mathit{false}) \subseteq \mathcal{I}_S(\Gamma(\mathit{false})) = \mathcal{I}_S(\mathit{false})$.

$\Leftarrow$.If $\mathcal{I}_S(\mathit{false}) \neq \emptyset$, then by the definition of reachable set there must be a Horn clause $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow \mathit{false}$ such that there exist tuples $\mathbf{t}_1, \ldots, \mathbf{t}_n$ satisfying $\mathbf{t}_i \in \mathcal{I}_S(p_i)$ for each $i \in \{1, \ldots, n\}$ and $(\mathbf{t}_1, \ldots, \mathbf{t}_n, \epsilon) \in I(C)$. We replace each uninterpreted predicate p_i by a fresh symbol p'_i, obtaining a Horn clause $C \wedge p'_1(\mathbf{x}_1) \wedge \cdots \wedge p'_n(\mathbf{x}_n) \rightarrow \mathit{false}$.

By the definition of reachable set, for each $\mathbf{t}_i$ there must be a Horn clause $C_i \wedge B_{i1} \wedge \cdots \wedge B_{in_i} \rightarrow H_i$ in $\mathcal{HC}$, such that $H_i = p_i(\mathbf{x}_{ih})$ and there exist tuples $\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in_i}$ satisfying $\mathbf{t}_{ij} \in \mathcal{I}_S(p_{ij})$ for each $j = \{1, \ldots, n_i\}$ and $(\mathbf{t}_{i1}, \ldots, \mathbf{t}_{in_i}, \mathbf{t}_i) \in I(C_i)$. We replace each uninterpreted predicate p_{ij} by a fresh symbol p'_{ij} for $j \in \{1, \ldots, n_i\}$, and replace p_i in H_i by p'_i, obtaining a Horn clause $C_i \wedge p'_{i1}(\mathbf{x}_{i1}) \wedge \cdots \wedge p'_{in_i}(\mathbf{x}_{in_i}) \rightarrow p'_i(\mathbf{x}_{ih})$.

By repeating this reasoning we can collect a set of Horn clauses $\mathcal{HC}_2$, which is in fact an unfolding of $\mathcal{HC}$. We denote the reachable set of $\mathcal{HC}_2$ as $\mathcal{I}^2_S$. By the definition of *reachable set* and the way in which $\mathcal{HC}_2$ is obtained, we know that $\epsilon \in \mathcal{I}^2_S(\mathit{false})$, so $\mathcal{HC}_2$ is unsolvable.

Theorem 1 follows from Lemma 1 and Lemma 2. Intuitively, we can prove the semantic solvability of a set of Horn clauses by proving that all of its unfoldings are semantically solvable. In the following section, we will describe an algorithm for Horn clause solving, which follows this idea.

5 Trace Abstraction Refinement for Horn Clause Solving

5.1 Trace abstraction refinement

We define the tree language corresponding to a set $\mathcal{HC}$ of Horn clauses on the alphabet $\mathcal{HC}$, i.e., the set of Horn clauses itself. For each Horn clause h: $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$, we define the arity of h to be n.

The tree automaton $\mathcal{A}_{\mathcal{HC}}$ for a set $\mathcal{HC}$ of Horn clauses is the tuple (Q, Σ, Q_f, Δ) such that

- $Q = \mathcal{R} \cup \{false\}$,
- $\Sigma = \mathcal{HC}$,
- $Q_f = \{false\}$,
- $\Delta = \{h(q_1, \ldots, q_n) = q_h | h : C \wedge q_1(\mathbf{x}_1) \wedge \cdots \wedge q_n(\mathbf{x}_n) \rightarrow q_h(\mathbf{x}_h) \in \mathcal{HC}\}$.

Example 2. For the set $\mathcal{HC}_1$ of Horn clauses given in Example 1, the set of states of the corresponding tree automaton $\mathcal{A}_{\mathcal{HC}_1}$ is $Q^{\mathcal{HC}_1} = \{false, p, q\}$, the alphabet of the tree language is $\Sigma^{\mathcal{HC}_1} = \{h1, h2, h3, h4\}$, the set of final states is $Q_f^{\mathcal{HC}_1} = \{false\}$ and the transition relation is: $\{h1() = p, h2(p) = p, h3(p) = q, h4(q) = false\}$. Moreover, the ground unfolding $\mathcal{HC}_1'$ given in Example 1 corresponds to the tree $h4(h3(h1, h2(h1)))$ (Figure 2b), which is accepted by $\mathcal{A}_{\mathcal{HC}_1}$.

Each tree accepted by $\mathcal{A}_{\mathcal{HC}}$ corresponds to a ground unfolding $\mathcal{HC}'$ of $\mathcal{HC}$: each node of the tree corresponds to a Horn clause $h' \in \mathcal{HC}'$, and vice versa. Moreover, if the node is labeled with a Horn clause $h \in \mathcal{HC}$, then it must be the case that $\Gamma(h') = h$. Since there is a one-to-one correspondence between trees and unfoldings, in the sequel, we will abuse the term of an (accepted) *tree* and a (ground) *unfolding*. Combining Theorem 1, in order to prove that $\mathcal{HC}$ is semantically solvable, one just needs to prove that all the trees accepted by $\mathcal{A}_{\mathcal{HC}}$ are semantically solvable.

Sets of solvable ground unfoldings could be represented by tree automata, and the semantic solvability verification task could be accomplished by constructing and manipulating tree automata, which is in spirit similar to the method in [4].

For finite automata, a *trace* is a sequence, while for tree automata, a *trace* is a tree. Similar to [4], we define *trace abstractions* for sets of Horn clauses.

Definition 4 (Trace abstraction). *A* trace abstraction *of a set of Horn clauses $\mathcal{HC}$ is a tuple of tree automata $(\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_n)$ such that each $\mathcal{A}_i$ $(i \in \{1, \ldots, n\})$ accepts a set of semantically solvable ground unfoldings of $\mathcal{HC}$.*

Let us denote by $L(\mathcal{A})$ the language accepted by $\mathcal{A}$. For each unfolding $\mathcal{HC}'$, we denote by $t_{\mathcal{HC}'}$ the tree corresponding to $\mathcal{HC}'$.

Combining Theorem 1 and Definition 4, the following theorem can be easily obtained.

Theorem 2. *Given a set of Horn clauses $\mathcal{HC}$, if there is a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ such that for each ground unfolding $\mathcal{HC}'$ of $\mathcal{HC}$, there is an $i \in \{1, \ldots, n\}$ such that $t_{\mathcal{HC}'} \in L(\mathcal{A}_i)$, then $\mathcal{HC}$ is semantically solvable.*

Based on the above discussions, we present a semi-algorithm for the verification of Horn clause semantic solvability, which is shown in Algorithm 1. It incrementally builds a trace abstraction $(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ such that $L(\mathcal{A}_{\mathcal{HC}}) \subseteq L(\mathcal{A}_1) \cup \cdots \cup L(\mathcal{A}_n)$. In the k'th iteration of the algorithm a tree $t \in L(\mathcal{A}_{\mathcal{HC}}) \backslash (L(\mathcal{A}_1) \cup \cdots \cup L(\mathcal{A}_{k-1}))$ is extracted and checked whether it is solvable. If t is not solvable, then $\mathcal{HC}$ is not solvable and the algorithm terminates and returns *false*, otherwise a set of trees, including t, is proved to be semantically solvable, and a tree automaton $\mathcal{A}_k$ accepting this set of trees is constructed and added to the trace abstraction. The algorithm could either terminate or loop forever. When it terminates, a result is given indicating whether $\mathcal{HC}$ is semantically solvable. A solvable ground unfolding corresponds to a spurious counterexample in CEGAR procedure for program verification, while an unsolvable ground unfolding corresponds to a concrete counterexample.

As mentioned previously, tree automata are closed under intersection, union, and complement. The task of checking whether there is a tree t in $L(\mathcal{A}_{\mathcal{HC}}) \backslash (L(\mathcal{A}_1) \cup \cdots \cup L(\mathcal{A}_{k-1}))$ for the series of tree automata $\mathcal{A}_{\mathcal{HC}}, \mathcal{A}_1, \ldots, \mathcal{A}_{k-1}$ can be done using existing algorithms on tree automata.

The task of checking whether a ground unfolding is solvable (line 5 of Algorithm 1) and constructing tree automata which exclude solvable ground unfoldings (line 8 of Algorithm 1) will be explained in the next subsection.

Algorithm 1 The trace abstraction refinement procedure

Input $\mathcal{HC}$: the set of Horn clauses
Output *true* if $\mathcal{HC}$ is semantically solvable, and *false* otherwise
1: Construct the tree automaton $\mathcal{A}_{\mathcal{HC}}$
2: $k \leftarrow 1$
3: **while** $\exists t \in L(\mathcal{A}_{\mathcal{HC}}) \backslash (L(\mathcal{A}_1) \cup \cdots \cup L(\mathcal{A}_{k-1}))$ **do**
4: Let $\mathcal{HC}'$ be the unfolding of $\mathcal{HC}$ induced by t
5: **if** $\phi_{constr}(\mathcal{HC}')$ is satisfiable **then**
6: **return** *false*
7: **else**
8: $\mathcal{A}_k \leftarrow$ ConstructTreeAutomaton(t, $\mathcal{HC}$)
9: **end if**
10: $k \leftarrow k + 1$
11: **end while**
12: **return** *true*

5.2 Interpolant tree automata

Given a ground unfolding $\mathcal{HC}'$, we can obtain a tree of formulas by extracting the constraint formulas in each Horn clause and renaming the variables. We define *enc* which encodes a ground unfolding to a tree of formulas as follows:

- suppose the Horn clause in $\mathcal{HC}'$ with *false* in its head is $C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow \mathit{false}$, $enc(\mathcal{HC}')$ is a tree that has $C[\mathbf{x}_0/\mathbf{x}'_0, \ldots, \mathbf{x}_n/\mathbf{x}'_n]$ in its root

node and $enc(hc_1, \mathbf{x}'_1), \ldots, enc(hc_n, \mathbf{x}'_n)$ as its subtrees, where $hc_i \in \mathcal{HC}'$ is the Horn clause with the uninterpreted predicate p_i in its head;
- for each Horn clause $hc : C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow p_h(\mathbf{x}_h) \in \mathcal{HC}'$ and a tuple of variables $\mathbf{x}^1_h$, $enc(hc, \mathbf{x}^1_h)$ is a tree that has $C[\mathbf{x}_0/\mathbf{x}'_0, \ldots, \mathbf{x}_n/\mathbf{x}'_n, \mathbf{x}_h/\mathbf{x}^1_h]$ in its root node and $enc(hc_1, \mathbf{x}'_1), \ldots, enc(hc_n, \mathbf{x}'_n)$ as its subtrees, where $hc_i \in \mathcal{HC}'$ is the Horn clause with the uninterpreted predicate p_i in its head;

where $\mathbf{x}'_i$ stands for a tuple of fresh variables corresponding to $\mathbf{x}_i$, and $C[\mathbf{x}_0/\mathbf{x}'_0, \ldots, \mathbf{x}_k/\mathbf{x}'_k]$ is the formula obtained by replacing in C the tuples of free variables $\mathbf{x}_0, \ldots, \mathbf{x}_k$ with $\mathbf{x}'_0, \ldots, \mathbf{x}'_k$.

We denote $\phi_{constr}(\mathcal{HC}')$ of an unfolding $\mathcal{HC}'$ as the conjunction of all the formulas in the tree $enc(\mathcal{HC}')$:

$$\phi_{constr}(\mathcal{HC}') = \bigwedge_{\{\phi | \phi \text{ is a formula in } enc(\mathcal{HC}')\}} \phi,$$

then we can prove that $\mathcal{HC}'$ is semantically solvable iff $\phi_{constr}(\mathcal{HC}')$ is unsatisfiable, and the latter can be checked by SMT solvers. Thus we have line 5 in Algorithm 1.

According to existing works such as [11], [9] and [7], interpolation techniques can be used to obtain a syntactic solution for a solvable ground unfolding. If $\phi_{constr}(\mathcal{HC}')$ is unsatisfiable, we can get a *tree interpolant* for $enc(\mathcal{HC}')$. A *tree interpolant* for a tree of formulas is a mapping TI from each tree node to a formula such that for each node n in the tree:

- $\bigwedge_{i \in children(n)} TI(i) \wedge \phi(n)$ implies $TI(n)$, and
- the set of variables appearing in $TI(n)$ is the intersection of the set of variables appearing in the formulas of the subtree rooted at n and the remaining formulas in the tree other than the subtree, and
- for the root node r, $TI(r) = \mathit{false}$,

where $children(n)$ is the set of child nodes of n, and $\phi(n)$ is the formula in the node n. Obviously, each node of $enc(\mathcal{HC}')$ corresponds to an uninterpreted predicate in $\mathcal{HC}'$, where the root of the subtree $enc(hc, \mathbf{x}^1_h)$ corresponds to the uninterpreted predicate appearing in the head of hc, and the root of $enc(\mathcal{HC}')$ corresponds to the special uninterpreted predicate *false*. Based on this correspondence, a syntactic solution of $\mathcal{HC}'$ can be obtained from a tree interpolant of $enc(\mathcal{HC}')$ up to variable renaming.

Example 3. From the ground unfolding $\mathcal{HC}'_1$ in Example 1 we can obtain a tree $enc(\mathcal{HC}'_1)$ of formulas shown in Figure 3a. Moreover, we have:

$$\phi_{constr}(\mathcal{HC}'_1) = x_1 > x_2 \wedge true \wedge x_1 = x_3 \wedge x_2 = x_4 + 1 \wedge x_3 = x_4,$$

which is unsatisfiable. A possible tree interpolant TI of this tree of formulas is shown in Figure 3b, where the formulas are positioned corresponding to the nodes of the tree in Figure 3a. The uninterpreted predicates (along with the parameter variables) corresponding to each node of $enc(\mathcal{HC}'_1)$ is shown in Figure 3c. According to this correspondence, a syntactic solution Π_{TI} of $\mathcal{HC}'_1$ can

be obtained from TI:

$\Pi_{TI}(\mathit{false}) = \mathit{false}$
$\Pi_{TI}(q_1)(x_1, x_2) = x_1 \leq x_2$
$\Pi_{TI}(p_1)(x_1, x_3) = x_1 \leq x_3$
$\Pi_{TI}(p_2)(x_3, x_2) = x_3 \leq x_2$
$\Pi_{TI}(p_3)(x_3, x_4) = x_3 \leq x_4$

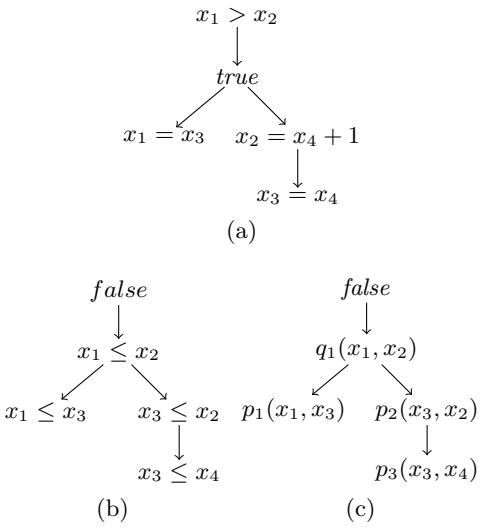

Fig. 3: (3a) the tree of formulas $enc(\mathcal{HC}'_1)$ for the ground unfolding $\mathcal{HC}'_1$ in Example 1, (3b) a tree interpolant TI for $enc(\mathcal{HC}'_1)$ in Figure 3a, and (3c) the uninterpreted predicates (along with parameter variables) corresponding to each node of $enc(\mathcal{HC}'_1)$.

Tree interpolation is an extension of Craig interpolation, and can be obtained from a tree of formulas using some of the existing interpolating SMT solvers such as Z3 [23] and SMTInterpol [24]. Based on the Craig interpolation theorem [25], $\phi_{constr}(\mathcal{HC}')$ is unsatisfiable iff there is a tree interpolant TI for $enc(\mathcal{HC}')$. In the sequel, we denote as Π_{TI} the syntactic solution of the unfolding obtained from the tree interpolant TI.

Given a solvable ground unfolding $\mathcal{HC}'$ and a corresponding tree interpolant, a tree automaton can be constructed, which accepts $\mathcal{HC}'$ and possibly other solvable ground unfoldings.

Definition 5 (Interpolant tree automata). *Given a solvable unfolding $\mathcal{HC}'$ of $\mathcal{HC}$ and a tree interpolant TI of $enc(\mathcal{HC}')$, an* interpolant tree automaton *is a tree automaton $(Q', \Sigma', Q'_f, \Delta')$ such that*

- $Q' = \mathcal{R}_{\mathcal{HC}'} \cup \{\text{false}\}$,

- $\Sigma' = \mathcal{HC}$,
- $Q'_f = \{\text{false}\}$,
- $h(p'_1, \ldots, p'_n) = p'_h \in \Delta'$ *implies*
  - h *is a Horn clause* $C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow p_h(\mathbf{x}_h)$ *in* $\mathcal{HC}$, *and*
  - $\Gamma(p'_i) = p_i$ *for* $i \in \{1, \ldots, n\} \cup \{h\}$, *and*
  - $(U, I) \models Cl_\forall(C \wedge \Pi_{TI}(p'_1)(\mathbf{x}_1) \wedge \Pi_{TI}(p'_2)(\mathbf{x}_2) \wedge \cdots \wedge \Pi_{TI}(p'_n)(\mathbf{x}_n) \rightarrow \Pi_{TI}(p'_h))(\mathbf{x}_h)$,
- $h(p'_1, \ldots, p'_n) = p'_h \in \Delta'$, *if there is a Horn clause* $h' : C \wedge p'_1(\mathbf{x}_1) \wedge \cdots \wedge p'_n(\mathbf{x}_n) \rightarrow p'_h(\mathbf{x}_h)$ *in* $\mathcal{HC}'$, *such that* $\Gamma(h') = h$,

where p'_h *and* p_h *might be the special uninterpreted predicate* false.

The alphabet of the interpolant tree automaton is the original set of Horn clauses, while the set of states is the set of uninterpreted predicates in the unfolding. In fact, the alphabet of all the tree automata constructed during the trace abstraction refinement procedure are the original set of Horn clauses. We can check that for each solvable ground unfolding $\mathcal{HC}'$, the corresponding interpolant tree automaton accepts $\mathcal{HC}'$. According to Definition 5, there are many interpolant tree automata for a ground unfolding and a corresponding tree interpolant. One can vary by deciding which transition rules should be added to the automata and which rules should be ignored. Here we give a *canonical scheme*, which adds all the possible transition rules to the interpolant tree automata.

Definition 6 (Canonical interpolant tree automata). *Given a solvable unfolding* $\mathcal{HC}'$ *of* $\mathcal{HC}$ *and a tree interpolant* TI *of* $enc(\mathcal{HC}')$, *the* canonical *interpolant tree automaton is an interpolant tree automaton* $(Q', \Sigma', Q'_f, \Delta')$ *such that*

- $h(p'_1, \ldots, p'_n) = p'_h \in \Delta'$ *iff*
  - h *is a Horn clause* $C \wedge p_1(\mathbf{x}_1) \wedge \cdots \wedge p_n(\mathbf{x}_n) \rightarrow p_h(\mathbf{x}_h)$ *in* $\mathcal{HC}$, *and*
  - $\Gamma(p'_i) = p_i$ *for* $i \in \{1, \ldots, n\} \cup \{h\}$, *and*
  - $(U, I) \models Cl_\forall(C \wedge \Pi_{TI}(p'_1)(\mathbf{x}_1) \wedge \Pi_{TI}(p'_2)(\mathbf{x}_2) \wedge \cdots \wedge \Pi_{TI}(p'_n)(\mathbf{x}_n) \rightarrow \Pi_{TI}(p'_h))(\mathbf{x}_h)$,

where p_h *and* p'_h *might be the special uninterpreted predicate* false.

A canonical interpolant tree automaton of a ground unfolding $\mathcal{HC}'$ typically accepts more ground unfoldings than just $\mathcal{HC}'$. We can prove that, each ground unfolding accepted by an interpolant tree automaton is syntactically solvable, which is stated in the following theorem.

Theorem 3. *For each tree* t *accepted by an interpolant tree automaton* $\mathcal{A}$, *the ground unfolding* $\mathcal{HC}_t$ *corresponding to* t *is syntactically solvable.*

Proof. By Definition 5, there is a tree interpolant TI, according to which $\mathcal{A}$ is constructed. Consequently there is a Π_{TI} which maps each state of $\mathcal{A}$ to a formula in TI. For an arbitrary uninterpreted predicate p_t that appeared in $\mathcal{HC}_t$, there is exactly one Horn clause $h_t \in \mathcal{HC}_t$ containing p_t in its head, suppose the sub-tree with $\Gamma(h_t)$ as the root is recognized at q_{h_t} in $\mathcal{A}$. We define a map Π_t as: $\Pi_t(p_t) = \Pi_{TI}(q_{h_t})$. It can be checked that Π_t is a syntactic solution of $\mathcal{HC}_t$ up to variable renaming.

Now we can describe an implementation of the function ConstructTreeAutomaton. Given a tree t and a set of Horn clauses $\mathcal{HC}$, it first obtains the ground unfolding $\mathcal{HC}'$ of $\mathcal{HC}$ induced by t, and then uses an interpolating SMT solver to compute a tree interpolant, based on which a canonical interpolant tree automaton is constructed. This implementation is given in Algorithm 2.

Algorithm 2 ConstructTreeAutomaton

Input $\mathcal{HC}$: the set of Horn clauses, t: a solvable tree accepted by $\mathcal{A}_{\mathcal{HC}}$
Output $\mathcal{A}_t$: a tree automaton accepting t and possibly other solvable trees
1: Let $\mathcal{HC}'$ be the unfolding of $\mathcal{HC}$ induced by t
2: Use SMT Interpolater to get the tree interpolant TI for $\mathcal{HC}'$
3: Construct the canonical interpolant tree automaton $\mathcal{A}_t$ according to Definition 6
4: **return** $\mathcal{A}_t$

Example 4. For the ground unfolding in Example 1, which corresponds to the tree $h4(h3(h1, h2(h1)))$, a tree interpolant TI is given in Example 3 along with the corresponding syntactic solution Π_{TI}. According to Π_{TI}, we can construct a canonical interpolant tree automaton $\mathcal{A}' = (Q', \Sigma', Q'_f, \Delta')$, where

- $Q' = \{false, q_1, p_1, p_2, p_3\}$,
- $\Sigma' = \{h1, h2, h3, h4\}$,
- $Q'_f = \{false\}$,
- $\Delta' = \{h4(q_1) = false\} \cup \{h3(p_i, p_j) = q_1 | i, j \in \{1, 2, 3\}\} \cup \{h2(p_i) = p_j | i, j \in \{1, 2, 3\}\} \cup \{h1() = p_i | i \in \{1, 2, 3\}\}$.

We can check that $h4(h3(h1, h2(h1)))$ is accepted by $\mathcal{A}'$. Moreover, $L(\mathcal{A}_{\mathcal{HC}_1}) = L(\mathcal{A}')$, so the set of Horn clauses $\mathcal{HC}_1$ has already been proven syntactically solvable, and $(\mathcal{A}')$ serves as the final trace abstraction.

If there is a series of interpolant tree automata covering all ground unfoldings of $\mathcal{HC}$, then $\mathcal{HC}$ is syntactically solvable. Moreover, we can obtain a syntactic solution from the series of interpolant tree automata.

Theorem 4. *For a set $\mathcal{HC}$ of Horn clauses, there is a series of interpolant tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ such that $\mathcal{A}_{\mathcal{HC}} \subseteq \mathcal{A}_1 \cup \cdots \cup \mathcal{A}_n$ iff the set of $\mathcal{HC}$ is syntactically solvable.*

Proof. $\Leftarrow$. If $\mathcal{HC}$ is syntactically solvable, then there is a syntactic solution Π_s. Similar to Definition 6, we could construct an interpolant automaton $\mathcal{A}_s$ using this solution, with $\mathcal{R}_{\mathcal{HC}} \cup \{false\}$ as the set of states. We can check that $L(\mathcal{A}_s) = L(\mathcal{A}_{\mathcal{HC}})$, in fact $\mathcal{A}_s$ is exactly the same as $\mathcal{A}_{\mathcal{HC}}$, since Π_s satisfies every Horn clause in $\mathcal{HC}$. Thus $(\mathcal{A}_s)$ is a qualifying trace abstraction.

$\Rightarrow$. Suppose there is a series of interpolant tree automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ as described above, where $\mathcal{A}_i = (Q^i, \Sigma^i, Q^i_f, \Delta^i)$. we should construct a solution for $\mathcal{HC}$.

By Definition 5 each state s in Q^i for some $i \in \{1, \ldots, n\}$ corresponds to a formula, which we denote by $\Pi_I(s)$. We define the formula ϕ_ρ for a tree ρ as $\phi_\rho = \bigwedge_{\rho \to s} \Pi_I(s)$, where $\rho \to s$ means that ρ is accepted at s in $\mathcal{A}_i$ for some $i \in \{1, \ldots, n\}$.

Each state of $\mathcal{A}_{\mathcal{HC}}$ can be reached by (possibly infinitely) many trees in $\mathcal{A}_{\mathcal{HC}}$. We define $\Pi_S(p) = \bigvee_{\rho \to_{\mathcal{A}_{\mathcal{HC}}} p} \phi_\rho$, where $\rho \to_{\mathcal{A}_{\mathcal{HC}}} p$ means that ρ is accepted at p in $\mathcal{A}_{\mathcal{HC}}$. For two trees ρ_1 and ρ_2, we can see that $\phi_{\rho_1} = \phi_{\rho_2}$ if $\{s|\rho_1 \to s\} = \{s|\rho_2 \to s\}$. Since the number of states in $Q^1 \cup \cdots \cup Q^n$ is finite, there are finitely many distinct ϕ_ρ for all trees ρ satisfying $\rho \to_{\mathcal{A}_{\mathcal{HC}}} p$. So $\Pi_S(p)$ is a well-formed formula. It can be checked that Π_S is a syntactic solution for $\mathcal{HC}$.

Example 5. Using the construction method given in the above proof, we can construct a syntactic solution $\Pi_{\mathcal{HC}_1}$ for $\mathcal{HC}_1$ according to the interpolants in Example 4:

$\Pi_{\mathcal{HC}_1}(q)(x, y) = x \leq y$

$\Pi_{\mathcal{HC}_1}(p)(x, y) = x \leq y$

We can easily check that $\Pi_{\mathcal{HC}_1}$ is indeed a syntactic solution for $\Pi_{\mathcal{HC}_1}$.

By Theorem 4, we can immediately get the following corollary:

Corollary 1. *If Algorithm 1 terminates and returns true, then $\mathcal{HC}$ is syntactically solvable.*

5.3 Discussion

Algorithm 1 might not necessarily terminate. However, this is quite reasonable, since program verification (which can be reduced to Horn clause solving problems) is generally undecidable. Existing works on program verification usually put a (time, space, etc.) limit on the algorithm, and output "UNKNOWN" if the algorithm can not return a result under the limit. Our algorithm can also be modified like that.

Our method is one among the several Horn clause solving methods originating from program verification techniques. Similar to the other methods, our method also inherit the advantages and disadvantages of the corresponding program verification technique.

There are two potential overheads in our method. One is the SMT solving procedure, which is well known to be time consuming. The other is tree automata operations.

When a ground unfolding is found in each iteration, the SMT solver is called to judge whether the unfolding is solvable, and if the answer is yes, a tree interpolant is generated. A ground unfolding is a group of Horn clauses, the size of the formula to be solved is linear to the size of the ground unfolding.

In the tree automata construction step, each transition is added to the tree automata according to the result of solving a relatively small SMT formula, the size of which is linear to the size of a Horn clause. It is possible that by just adding a few transitions, a large amount of ground unfoldings are excluded from future

iterations. So the tree interpolant generated from the ground unfolding is largely reused to exclude many more ground unfoldings. Since the computation time of SMT solving is exponential to the size of the formula in the worst case, solving small SMT formulas spends much less time than solving large SMT formulas.

The process of finding new ground unfoldings that have not been excluded from the current trace abstraction involves tree automata operations. The potential overhead of our method lies partly in the tree automata operations. In the k'th iteration, we want to find a tree from the language of $\mathcal{A}_{\mathcal{HC}} \cap \overline{\mathcal{A}_1} \cap \cdots \overline{\mathcal{A}_{k-1}}$. The complement operation might lead to exponential blow up, and the task of checking emptiness of the intersection of a set of tree automata is EXPTIME-complete [22]. However, by using techniques such as binary decision diagrams (BDD), we might be able to handle tree automata of very large sizes.

For many other CEGAR-based verification methods, the construction of abstract models in each iteration involves a large number of SMT calls. In trace abstraction refinement, however, the number of states in each tree interpolant automaton is linear to the size of the ground unfolding, which is relatively small compared to the whole system. Consequently less time is spent on SMT solving in each iteration. However, the number of iterations might be large in some cases, and the tree automata operations become a possible bottleneck.

6 Program verification by solving Horn clauses

In Section 2, we have already shown that safety verification of simple programs can be easily encoded into Horn clause solving problems. In this section we describe existing works on how the safety verification of multi-threaded programs can be encoded into Horn clauses in a compositional way, and how the safety verification of programs with procedures can be encoded into Horn clauses. (These encoding schemes are described in [6].) By showing the above encoding schemes, we demonstrate that our work can be seen as an extension of the original trace abstraction refinement scheme for the verification of various programs.

6.1 Compositional verification of multi-threaded programs

Safety verification task of multi-threaded programs can be encoded into Horn clause solving problem using Owicki-Gries method [12]. For an N-thread program with the set V of variables, we use N uninterpreted predicates $Inv_1, \ldots, Inv_N$ over V to represent assertions for each thread. The Owicki-Gries proof rule is shown in Figure 4.

Each Inv_i represents certain invariant property of thread i. The CO1 rule says that every Inv_i holds initially. CO2 says that Inv_i is invariant with respect to transitions of thread i. CO3 says that each Inv_i is invariant with respect to transitions of any other thread $j \neq i$. CO4 says that the system is safe as long as Inv_i holds for each thread i.

According to Owicki-Gries rules, if we can find a formula for each Inv_i, $i \in \{1 \ldots N\}$, then the multi-threaded program can be proved to be safe. The

$$\begin{array}{ll}
\text{CO1: } init(V) & \rightarrow Inv_i(V) \\
\text{CO2: } Inv_i(V) \wedge \rho_i(V, V') & \rightarrow Inv_i(V') \\
\text{CO3: } Inv_i(V) \wedge (\bigvee_{j \in 1 \ldots N \setminus \{i\}} Inv_j(V) \wedge \rho_j(V, V')) & \rightarrow Inv_i(V') \\
\text{CO4: } Inv_1(V) \wedge \cdots \wedge Inv_N(V) \wedge unsafe(V) & \rightarrow false \\
\hline
\multicolumn{2}{c}{\text{the multi-threaded program is safe}}
\end{array}$$

Fig. 4: Owicki-Gries rules.

premise part of the Owicki-Gries rules is in fact a set of Horn clauses with uninterpreted predicates $Inv_1, \ldots, Inv_N$, and finding a formula for each Inv_i can be accomplished by obtaining a syntactic solution for this set of Horn clauses.

Example 6. For the simple multi-threaded program `LockBit` [11] in Figure 5, where two threads access critical regions exclusively, the safety property to be verified is that the two threads will never be in the critical section at the same time. The Horn clause encoding is shown in Figure 6, where Inv_1 and Inv_2 are two uninterpreted predicates, and $init$,ρ_1, ρ_2 and $unsafe$ are formulas in the constraint language described as follows:

- $init(pc_1, pc_2, lock) = (pc_1 = a \wedge pc_2 = p \wedge lock = 0)$,
- $\rho_1(pc_1, pc_2, lock, pc_1', pc_2', lock') = (lock = 0 \wedge lock' = 1 \wedge pc_1 = a \wedge pc_a' = b \wedge pc_2 = pc_2')$,
- $\rho_2(pc_1, pc_2, lock, pc_1', pc_2', lock') = (lock = 0 \wedge lock' = 1 \wedge pc_2 = p \wedge pc_2' = q \wedge pc_1 = pc_2')$,
- $unsafe(pc_1, pc_2, lock) = (pc_1 = b \wedge pc_2 = q)$.

The Horn clauses h_1^l and h_2^l are obtained according to CO1; h_3^l, h_4^l are obtained according to CO2; h_5^l, h_6^l are obtained according to CO3; h_7^l is obtained according to CO4.

When the verification task of a multi-threaded program is encoded into a Horn clause solving problem, the remaining work is left to the Horn clause solver. Thus, compositional verification is easily performed by a compositional encoding scheme.

```
// Thread 1                 // Thread 2
a: take_lock(lock, 1); p: take_lock(lock, 1);
b: //critical            q: //critical
```

Fig. 5: Multi-threaded program `LockBit`

6.2 Verification of programs with procedures

`McCarthy91` function: an example The safety verification tasks of recursive programs can also be encoded into Horn clause solving problems. Here we take the example of `McCarthy91` function.

$$(h_1^l)\ I_1(pc_1, pc_2, lock) \leftarrow init(pc_1, pc_2, lock)$$
$$(h_2^l)\ I_2(pc_1, pc_2, lock) \leftarrow init(pc_2, pc_2, lock)$$
$$(h_3^l)\ I_1(pc_1', pc_2', lock') \leftarrow I_1(pc_1, pc_2, lock) \wedge \rho_1(pc_1, pc_2, lock, pc_1', pc_2', lock')$$
$$(h_4^l)\ I_2(pc_1', pc_2', lock') \leftarrow I_2(pc_1, pc_2, lock) \wedge \rho_2(pc_1, pc_2, lock, pc_1', pc_2', lock')$$
$$(h_5^l)\ I_1(pc_1', pc_2', lock') \leftarrow I_2(pc_1, pc_2, lock) \wedge \rho_2(pc_1, pc_2, lock, pc_1', pc_2', lock')$$
$$(h_6^l)\ I_2(pc_1', pc_2', lock') \leftarrow I_1(pc_1, pc_2, lock) \wedge \rho_1(pc_1, pc_2, lock, pc_1', pc_2', lock')$$
$$(h_7^l)\ false \leftarrow I_1(pc_1, pc_2, lock) \wedge I_2(pc_1, pc_2, lock) \wedge unsafe(pc_1, pc_2, lock)$$

Fig. 6: Horn clause encoding for `LockBit`.

McCarthy 91 function

```
function McCarthy91 (x){
    if(x>100)return x-10;
    else return McCarthy91(McCarthy91(x+11));
}
assert(McCarthy91(x)>=91);
```

As shown in the assertion given above, we want to prove that for an arbitrary input x, the `McCarthy91` function always returns a value that is no less than 91.

We can encode this verification task into a set of Horn clauses as follows:

$$(h_1^m)\ M(x, y) \leftarrow x > 100 \wedge y = x - 10$$
$$(h_2^m)\ M(x, y) \leftarrow x \leq 100 \wedge u = x + 11 \wedge M(u, z) \wedge M(z, y)$$
$$(h_3^m)\ false \leftarrow M(x, y) \wedge y < 91$$

The corresponding tree automaton is $\mathcal{A}_m = (Q^m, Q_f^m, \Sigma^m, \Delta^m)$, where

- $Q^m = \{false, M\}$,
- $Q_f^m = \{false\}$,
- $\Sigma^m = \{h_1^m, h_2^m, h_3^m\}$,
- $\Delta^m = \{h_1^m() = M, h_2^m(M, M) = M, h_3^m(M) = false\}$.

We can find a tree $h_3^m(h_1^m)$ accepted by $\mathcal{A}_m$. The ground unfolding $\mathcal{HC}_{m1}$ corresponding to this tree is obtained as follows:

$$(h_1^{m1})\ false \leftarrow M_1(x, y) \wedge y < 91$$
$$(h_2^{m1})\ M_1(x, y) \leftarrow x > 100 \wedge y = x - 10$$

One possible syntactic solution Π_{m1} obtained from a tree interpolant of $enc(\mathcal{HC}_{m1})$ is $\Pi_{m1}(M_1)(x, y) : y \geq 91$. According to this tree interpolant, a tree automaton $\mathcal{A}_{m1} = (Q^{m1}, Q_f^{m1}, \Sigma^{m1}, \Delta^{m1})$ can be constructed, where

- $Q^{m1} = \{false, M_1\}$,
- $Q_f^{m1} = \{false\}$,
- $\Sigma^{m1} = \{h_1^m, h_2^m, h_3^m\}$,

– $\Delta^{m1} = \{h_1^m() = M_1, h_2^m(M_1, M_1) = M_1,$
$h_3^m(M_1) = \mathit{false}\}$.

One can check that $L(\mathcal{A}_{m1}) = L(\mathcal{A}_m)$, so we arrive to the conclusion that each tree accepted by $\mathcal{A}_m$ corresponds to a solvable unfolding, i.e., the property of the `McCarthy91` function mentioned previously is proved to be correct.

This might seem surprising at first sight: we only analyzed one tree $h_3^m(h_1^m)$, which only concerns the "if" branch (where $x > 100$) of the `McCarthy91` function, yet the function is proved to be correct by the single interpolant tree automaton constructed from $h_3^m(h_1^m)$. How could the function be proved to be correct, if the "then" branch (where $x \leq 100$) is not even considered? Moreover, if this proof is valid, then we can claim that the same property holds if we change "$x + 11$" in the program to "$x - 11$", which seems absurd, but is in fact not at all.

By a careful check one can find that this function returns a value only when the condition $x > 100$ for the input x is met, otherwise it calls itself again and again, and does not return. Therefore the solution of the Horn clause proves the partial correctness of this function: if the `McCarthy91` function ends and returns, the returned value must be no less than 91. Consequently, we can change the expression "$x + 11$" in the program to anything, and the property still holds. This surprisingly simple solution helps us to gain insight into the `McCarthy91` function.

From this example, we can see that the trace abstraction refinement successfully exploits the structure of the program to help the verification. This might be a potential advantage when dealing with some programs.

Encoding programs with procedures Generally, verification tasks of programs with procedures can be encoded into Horn clause solving problems according to the rules shown in Figure 7.

The entry point of the program is the procedure *main*. For each procedure f, V_f represents the set of global and local variables accessible in its scope. An uninterpreted predicate $T_f(V_f, V_f')$ is allocated for each procedure f. $T_f(V_f, V_f')$ is in fact a summary of f, which represents the reachability relation between the entry state V_f and the successor state V_f' at the same level of recursion. The rule CP1 says that the initial state is satisfied by the summary of *main*. CP2 says that the summary T_f is a closure of the internal transitions of f. CP3 says that if the caller procedure f satisfies the summary T_f at the calling point, then the callee procedure g satisfies T_g at the entry point. CP4 says that, if the callee procedure g satisfies the summary T_g between its entry point and return point, then T_f can be extended one step to include the call of g. Finally, CP5 says that the summary of f entails the safety of f.

7 Experiments

We have implemented our algorithm using SMTInterpol [24] as the interpolating SMT solver. We implemented a tree automata library, which represents tree

$$
\begin{array}{ll}
\text{CP1: } init(V_{main}) \wedge V_{main} = V'_{main} & \rightarrow T_{main}(V_{main}, V'_{main}) \\
\text{CP2: } T_f(V_f, V'_f) \wedge \rho_f(V'_f, V''_f) & \rightarrow T_f(V_f, V''_f) \\
\text{CP3: } T_f(V_f, V'_f) \wedge call_{f,g}(V'_f, V''_g) \wedge V''_g = V'''_g & \rightarrow T_g(V''_g, V'''_g) \\
\text{CP4: } T_f(V_f, V'_f) \wedge call_{f,g}(V'_f, V''_g) \wedge & \\
\quad T_g(V''_g, V'''_g) \wedge ret_{f,g}(V''_g, V'''_f, V''''_f) \wedge loc_f(V'_f, V''''_f) & \rightarrow T_f(V'_f, V''''_f) \\
\text{CP5: } T_f(V_f, V'_f) \wedge unsafe(V'_f) & \rightarrow false \\
\hline
\multicolumn{2}{c}{\text{the program is safe}}
\end{array}
$$

Fig. 7: Proof rules for programs with procedures [6].

automata explicitly, and performs tree automata difference operation using an on-the-fly algorithm which combines complement and intersection together. For comparison, we have implemented a predicate abstraction based CEGAR algorithm, which also uses SMTInterpol as the underlying interpolating SMT solver. We have also performed experiments using an existing tool Eldarica[3], which performs a variant of predicate abstraction based CEGAR algorithm for Horn clause verification. Eldarica uses Princess [4] as the underlying interpolating SMT solver to generate new predicates for CEGAR.

The experiments are performed on a set of Horn clause benchmarks[5] from the SV-COMP[6] repository. All of these benchmarks are in integer linear arithmetic. According to [7], these benchmarks are collected from various sources, such as recursive algorithms, benchmarks extracted from programs with singly-linked lists, VHDL models of circuits, verification conditions for programs with arrays, benchmarks from the NECLA static analysis suite, and C programs with asynchronous procedure calls. Some of the models only contain linear Horn clauses, while others contain non-linear Horn clauses. The results are shown in Figure **??**.

The performance of our implementation of predicate abstraction based CEGAR is roughly similar to Eldarica. This is because they use similar algorithms. However, our implementation scales not so well as Eldarica for the models on which more iterations are required. This is quite reasonable since our implementation is relatively naive, while Eldarica is a sophisticated tool. There are also a few models on which our implementation behaves much better than Eldarica, e.g., *brprt* and *asfifoStatus*. One possible reason is that the two implementations use different interpolating SMT solvers, which might result in different interpolants, and consequently lead to different performances. Moreover, the difference in the implementation details, such as how the models are pre-processed and stored, could also lead to different performance.

We can see that on almost all of the models the trace abstraction refinement algorithm needs more iterations to reach a conclusion. Intuitively, this is because that the interpolant tree automaton constructed in each iteration is "small", i.e., it recognizes a relatively small set of infeasible traces (i.e., solvable ground

[3] http://lara.epfl.ch/w/eldarica
[4] http://www.philipp.ruemmer.org/princess.shtml
[5] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica
[6] http://sv-comp.sosy-lab.org

Model	#p	#c	TAR		PA		Eld
			tm/s	#it	tm/s	#it	tm/s
AsynPrg							
f_rec(E,N)	136	188	to	1123	324.25	78	15.52
h1(E,N)	52	60	18.26	82	1.39	4	1.87
h1h2(E,N)	53	63	to	417	4.32	8	2.50
nch(C,N)	102	122	to	265	157.63	41	16.90
plb_simple(C,N)	104	124	to	506	0.81	2	1.71
plb_simple(E,N)	104	126	to	461	9.92	10	4.30
server.manual(C,L)	11	14	0.27	4	0.11	2	0.83
simple(E,N)	52	60	16.24	76	1.47	4	1.92
test0(C,N)	52	60	to	428	5.91	11	2.64
test0(E,N)	50	58	21.87	96	1.36	4	1.79
test1(C,N)	77	87	to	633	22.82	21	4.03
test1(E,N)	75	85	to	461	16.78	18	3.77
test2_1(E,N)	67	79	to	377	7.57	12	2.60
test2_2(E,N)	67	79	to	322	7.92	12	3.05
test2(C,N)	69	81	to	396	17.38	19	4.48
test4(C,N)	97	117	to	762	20.89	19	3.22
test4(E,N)	97	117	to	729	5.71	7	3.06
test_recursion(E,N)	210	288	to	243	to	38	42.87
wrpc.manual(C,L)	9	14	2.21	35	3.35	16	1.00
wrpc(E,N)	88	109	42.27	82	4.78	5	2.93
NECLA							
blast(C,L)	33	43	8.93	65	4.16	14	1.90
inf1(E,L)	19	26	0.51	6	0.47	4	1.35
inf4(E,L)	33	48	0.93	10	1.24	6	2.49
inf6(C,L)	23	28	1.96	33	2.75	14	1.67
inf8(C,L)	29	39	6.99	73	7.39	18	2.18
L2CA							
bubblesort(E,L)	674	792	0.33	1	0.49	1	5.46
insdel(E,L)	28	32	0.17	1	0.11	1	0.82
insertsort(E,L)	130	170	0.30	1	0.48	1	2.13
listcounter(C,L)	31	36	to	171	to	87	to
listcounter(E,L)	31	35	0.21	1	0.23	1	1.47
listreversal(C,L)	97	108	35.79	149	14.01	25	9.65
listreversal(E,L)	99	108	0.30	1	0.39	1	2.24
mergesort(E,L)	544	607	0.32	1	0.48	1	6.74
selectionsort(E,L)	401	460	0.27	1	0.29	1	6.12
SIL							
rotation_vc1(C,L)	13	59	4.38	91	2.15	9	1.62
rotation_vc2(C,L)	20	96	22.51	507	5.73	17	1.98
rotation_vc3(C,L)	20	96	0.04	1	0.50	1	1.49
rotation_vc1(E,L)	13	61	0.64	3	0.66	3	1.40
split_vc1(C,L)	32	191	74.62	1473	29.16	30	2.78
split_vc2(C,L)	29	149	53.41	1113	4.22	11	to
split_vc3(C,L)	29	149	0.01	1	0.70	1	2.01
split_vc1(E,L)	38	191	0.50	4	9.84	8	3.53

Model	#p	#c	TAR		PA		Eld
			tm/s	#it	tm/s	#it	tm/s
RECUR							
addition(C,N)	9	11	0.26	3	0.24	3	0.75
bfprt(C,N)	14	17	0.43	6	0.20	3	to
binarysearch(C,N)	13	17	0.36	5	0.15	2	0.78
buildheap(C,N)	11	14	mo	27	to	35	sto
countZero(C,N)	13	18	to	1401	1.52	10	1.32
floodfill(C,N)	14	18	mo	42	3.13	11	sto
half(C,N)	10	12	0.56	5	0.34	4	sto
identity(C,N)	12	16	7.67	34	0.32	4	1.07
mccarthy91(C,N)	9	12	0.57	7	0.27	4	0.91
mccarthy92(C,N)	9	12	0.89	10	0.41	4	4.43
merge-leq(C,N)	14	19	0.78	10	0.67	6	1.35
merge(C,N)	14	19	0.86	10	0.57	5	1.36
palindrome(C,N)	10	12	0.61	6	0.39	4	1.31
parity(C,N)	13	16	0.62	7	0.51	6	1.07
remainder(C,N)	9	11	1.50	17	2.23	9	1.38
running(C,N)	10	12	0.40	5	0.26	4	1.05
running-old(C,N)	9	11	0.01	1	0.05	1	0.33
triple(C,N)	11	14	0.86	6	0.68	5	2.10
VHDL							
asfifoFE(C,L)	67	5291	to	1325	to	42	to
asfifoStatus(C,L)	67	3218	0.05	1	0.00	1	68.72
counter(C,L)	6	14	1.05	41	0.45	8	0.73
register(C,L)	10	46	4.22	255	0.16	5	0.60
synlifo(C,L)	67	987	to	3056	61.95	103	9.67
McMill06							
anubhav(C,L)	37	35	1.72	9	0.82	3	1.33
copy1(E,L)	37	37	12.79	19	31.28	13	3.98
cousout(C,N)	37	37	to	118	to	40	sto
loop(E,L)	37	37	10.87	19	20.35	12	3.42
loop1(E,L)	37	37	10.42	19	24.44	13	3.77
scan2(E,N)	45	46	0.47	2	0.25	1	1.30
scan(E,N)	37	39	to	69	to	37	65.92
string_concat1(E,L)	51	64	to	265	to	49	to
string_concat(E,N)	49	53	to	125	to	38	to
string_copy(E,N)	40	44	to	60	to	31	sto
substring1(E,N)	57	72	2.84	5	0.48	1	1.95
substring(E,N)	47	49	0.55	1	0.42	1	1.61
MONNIAUX							
boustrophedon(C,L)	28	35	53.65	193	37.97	19	sto
boustrophedon.ex-pansed(C,L)	30	39	69.06	340	18.80	18	sto
gopan(C,L)	32	38	to	143	to	42	sto
halbwachs(C,L)	38	44	to	110	to	42	sto
rate_limiter(C,N)	29	37	49.96	130	9.13	10	2.31

Fig. 8: Experimental results of trace abstraction refinement (TAR) in comparison with our implementation of predicate abstraction based CEGAR (PA) and Eldarica (Eld). The first letter after each model name indicates whether the model is correct (C) or erroneous (E), and the second letter indicates whether the model contains only linear Horn clauses (L) or contains some non-linear Horn clauses (N). "#p" is the number of uninterpreted predicates appearing in the model, and "#c" is the number of clauses contained in the model. "tm/s" is the total running time (in seconds), and "#it" is the number of iterations. "to" means timeout (500s), "mo" means memory out, and "sto" means stack overflow. The experiments are performed on a Intel Core i7 Duo CPU with 3.06GHz. The memory limit is set to 1000 MB.

unfoldings). As a result, the refinement steps are less progressive, and more iterations are needed for the algorithm to terminate. In Algorithm 2, each state of the interpolant tree automata is associated with only one formula from the tree interpolant, while in predicate abstraction based CEGAR algorithms, each abstract state is typically associated with a conjunction of one or more formulas from the tree interpolants. Since the interpolant tree automata are constructed according to the formulas from the tree interpolants, this lack of "combination" of formulas in the trace abstraction refinement algorithm could result in "small" interpolant tree automata.

On the other hand, according to Algorithm 2, the number of states of each interpolant tree automaton is bounded by the number of formulas in the corresponding tree interpolant, which is relatively small compared to the state space of the model. From the experimental results we can see that this almost always leads to faster iterations than that in predicate abstraction based CEGAR algorithms. In a sense, the trace abstraction refinement algorithm achieves faster iterations at the cost of less progressive refinement steps (and consequently larger numbers of iterations).

On many of the models, the performance of trace abstraction refinement is comparable to predicate abstraction based CEGAR, while on others it is significantly worse. Overally, the trace abstraction refinement performs not so well as predicate abstraction based CEGAR in our experiments. This is especially apparent for the *AsynPrg* benchmarks, where the trace abstraction refinement algorithm always performs large numbers of iterations while still not reaching a conclusion before timeout. According to the previous analysis, there is a trade-off between spending less time in each iteration and taking fewer iterations to reach a conclusion, and trace abstraction refinement and predicate abstraction based CEGAR can be seen as two extremes of the trade off. A possible way to improve the trace abstraction refinement algorithm is to compromise between these two extremes and combine the tree interpolants of multiple ground unfoldings for the refinement in each iteration. Techniques such as disjunctive interpolation [7] could be considered for this purpose. The resulting algorithm will take fewer iterations to reach a conclusion at the cost of more time spent in each iteration, possibly having better overall performance.

For *buildheap* and *floodfill*, the tree automata operations lead to state explosion, which result in exhausted memory. This gives us a hint to explore more efficient methods to store and manipulate tree automata.

8 Conclusion

In this paper we described a trace abstraction refinement scheme for Horn clause verification. The original trace abstraction refinement scheme used finite automata as the abstraction formalism and can only be used for verifying simple programs. We adapt it for verifying Horn clause solvability. Considering the flexibility of Horn clauses as intermediate languages in verification, our work makes it convenient to use the trace abstraction refinement scheme for various verifica-

tion tasks (e.g., the verification of multithreaded programs and programs with procedure calls). From another point of view, we provide a new method for solving sets of Horn clauses. This method exploits the structure information of Horn clauses by constructing and manipulating tree automata.

The experiments show that our method needs more iterations to terminate than predicate abstraction based CEGAR, while each iteration costs less time. We analyzed the difference between predicate abstraction based CEGAR and trace abstraction refinement and suggested a possible compromise between the two algorithms, which we will try in the future.

Our implementation is quite prototypical. We are considering improving the tree automata library by investigating and trying algorithms such as [26] and [21]. Moreover, preprocessing of Horn clauses using large block encoding [27] seems a promising improvement, since it might largely reduce the number of Horn clauses and uninterpreted predicates, resulting in smaller tree automata in the trace abstraction refinement procedure.

References

1. Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000) Counterexample-guided abstraction refinement. *Proc. of CAV 2000*, Chicago, IL, 15-19 July, pp. 154–169. Springer-Verlag, Berlin.
2. Henzinger, T. A., Jhala, R., Majumdar, R., and Sutre, G. (2002) Lazy abstraction. *Proc. of POPL 2002*, Portland, OR, 16-18 January, pp. 58–70. ACM, New York.
3. McMillan, K. L. (2006) Lazy abstraction with interpolants. *Proc. of CAV 2006*, Seattle, WA, 17-20 August, pp. 123–136. Springer-Verlag, Berlin.
4. Heizmann, M., Hoenicke, J., and Podelski, A. (2009) Refinement of trace abstraction. *Proc. of SAS 2009*, Los Angeles, CA, 9-11 August, pp. 69–85. Springer-Verlag, Berlin.
5. Jaffar, J. and Maher, M. J. (1994) Constraint logic programming: A survey. *J. Log. Program.*, **19–20**, 503–581.
6. Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. (2012) Synthesizing software verifiers from proof rules. *Proc. of PLDI 2012*, Beijing, China, 11-16 June, pp. 405–416. ACM, New York.
7. Rümmer, P., Hojjat, H., and Kuncak, V. (2013) Disjunctive interpolants for Horn-clause verification. *Proc. of CAV 2013*, Saint Petersburg, Russia, 13-19 July, pp. 347–363. Springer-Verlag, Berlin.
8. Hoder, K. and Bjørner, N. (2012) Generalized property directed reachability. *Proc. of SAT 2012*, Trento, Italy, 17-20 June, pp. 157–171. Springer-Verlag, Berlin.
9. McMillan, K. L. and Rybalchenko, A. (2013) Solving constrained Horn clauses using interpolation. Technical report. Technical Report MSR-TR-2013-6, Microsoft Research.
10. Bjørner, N., McMillan, K. L., and Rybalchenko, A. (2013) On solving universally quantified Horn clauses. *Proc. of SAS 2013*, Seattle, WA, 20-22 June, pp. 105–125. Springer-Verlag, Berlin.
11. Gupta, A., Popeea, C., and Rybalchenko, A. (2011) Predicate abstraction and refinement for verifying multi-threaded programs. *Proc. of POPL 2011*, Austin, Texas, USA, 26-28 January, pp. 331–344. ACM, New York.

12. Owicki, S. and Gries, D. (1976) An axiomatic proof technique for parallel programs i. *Acta Inf.*, **6**, 319–340.
13. Jones, C. B. (1983) Tentative steps toward a development method for interfering programs. *ACM Trans. Program. Lang. Syst.*, **5**, 596–619.
14. Heizmann, M., Hoenicke, J., and Podelski, A. (2013) Software model checking for people who love automata. *Proc. of CAV 2013*, Saint Petersburg, Russia, 13-19 July, pp. 36–52. Springer-Verlag, Berlin.
15. Heizmann, M., Hoenicke, J., and Podelski, A. (2010) Nested interpolants. *Proc. of POPL 2010*, Madrid, Spain, 17-23 January, pp. 471–482. ACM, New York.
16. Cassez, F., Müller, C., and Burnett, K. (2014) Summary-based inter-procedural analysis via modular trace refinement. *Proc. of FSTTCS 2014*, New Delhi, India, 15-17 December, pp. 545–556. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
17. Farzan, A., Kincaid, Z., and Podelski, A. (2014) Proofs that count. *Proc. of POPL 2014*, San Diego, CA, 20-21 January, pp. 151–164. ACM, New York.
18. Vizel, Y. and Gurfinkel, A. (2014) Interpolating property directed reachability. *Proc. of CAV 2014*, Vienna, Austria, 18-22 July, pp. 260–276. Springer-Verlag, Berlin.
19. Bradley, A. R. and Manna, Z. (2007) Checking safety by inductive generalization of counterexamples to induction. *Proc. of FMCAD 2007*, Austin, TX, 11-14 November, pp. 173–180. IEEE.
20. Kafle, B. and Gallagher, J. P. (2015) Tree automata-based refinement with application to Horn clause verification. *Proc. of VMCAI 2015*, Mumbai, India, 12-14 January, pp. 209–226. Springer-Verlag, Berlin.
21. Gallagher, J. P., Ajspur, M., and Kafle, B. (2014) An optimised algorithm for determinisation and completion of finite tree automata. Technical report. Technical Report 145, Roskilde University, Denmark.
22. Gécseg, F. and Steinby, M. (1997) Tree languages. In Rozenberg, G. and Salomaa, A. (eds.), *Handbook of Formal Languages*, pp. 1–68. Springer-Verlag, Berlin, DE.
23. De Moura, L. and Bjørner, N. (2008) Z3: An efficient SMT solver. *Proc. of TACAS 2008*, Budapest, Hungary, 29 March-6 April, pp. 337–340. Springer-Verlag, Berlin.
24. Christ, J., Hoenicke, J., and Nutz, A. (2012) SMTInterpol: An interpolating SMT solver. *Proc. of SPIN 2012*, Oxford, UK, 23-24 July, pp. 248–254. Springer-Verlag, Berlin.
25. Boolos, G. S., Burgess, J. P., and Jeffrey, R. C. (2003) *Computability and Logic*. Cambridge University Press, Cambridge.
26. Lengál, O., Šimáček, J., and Vojnar, T. (2012) Vata: A library for efficient manipulation of non-deterministic tree automata. *Proc. of TACAS 2012*, Tallinn, Estonia, 24 March-1 April, pp. 79–94. Springer-Verlag, Berlin.
27. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M. E., and Sebastiani, R. (2009) Software model checking via large-block encoding. *Proc. of FMCAD 2009*, Austin, Texas, USA, 15-18 November, pp. 25–32. IEEE.