In this paper we show that checking satisfiability of a set of non-linear Horn clauses (also called a non-linear Horn clause program) can be achieved using a solver for linear Horn clauses. We achieve this by interleaving a program transformation with a satisfiability checker for linear Horn clauses (also called a solver for linear Horn clauses). The program transformation is based on the notion of tree dimension, which we apply to a set of non-linear clauses, yielding a set whose derivation trees have bounded dimension. Such a set of clauses can be linearised. The main algorithm then proceeds by applying the linearisation transformation and solver for linear Horn clauses to a sequence of sets of clauses with successively increasing dimension bound. The approach is then further developed by using a solution of clauses of lower dimension to (partially) linearise clauses of higher dimension.

We constructed a prototype implementation of this approach and performed some experiments on a set of verification problems, which shows some promise.

1 Introduction

Many software verification problems can be reduced to checking satisfiability of a set of Horn clauses (the verification conditions). In this paper we propose an approach for checking satisfiability of a set of non-linear Horn clauses (clauses whose body contains more than one non-constraint atom) using a linear Horn clause solver. A program transformation based on the notion of tree dimension is applied to a set of non-linear Horn clauses; this gives a set of clauses that can be linearised and then solved using a linear solver for Horn clauses. This combination of dimension-bounding, linearisation and then solving with a linear solver is repeated for successively increasing dimension. The dimension of a tree is a measure of its non-linearity – for example a linear tree (whose nodes have at most one child) has dimension zero while a complete binary tree has dimension equal to its height.

A given set of Horn clauses $P$ can be transformed into a new set of clauses $P^{\leq k}$, whose derivation trees are the subset of $P$'s derivation trees with dimension at most $k$. It is known that $P^{\leq k}$ can be transformed to a linear set of clauses preserving satisfiability; hence if we can find a model of the linear set of clauses then the original clauses $P^{\leq k}$ also have a model.

The algorithm terminates with success if a model (solution) $M$ of $P^{\leq k}$ is also a model (after appropriate translation of predicate names) of $P$. However if $M$ is not a solution of $P$, then we proceed to generate $P^{\leq k+1}$ and repeat the procedure. The algorithm terminates with failure if $P^{\leq k}$ is shown to be unsatisfiable for some $k$, since this implies that $P$ is also unsatisfiable.
Solving non-linear Horn clauses using a linear Horn clause solver

A more sophisticated version of the algorithm attempts to use the model $M$ of $P^{\leq k}$ to (partially) linearise $P^{\leq k+1}$. We can exploit the model of $P^{\leq k}$ in the following way; if $P^{\leq k+1}$ has a counterexample that does not use the (approximate) solution $M$ for $P^{\leq k}$, then $P$ is unsatisfiable. We continue this process successively for increasing value of $k$ until we find a solution or a counterexample to $P$, or until resources are exhausted.

As an example program, we consider a set of constrained Horn clauses $P$ in Figure 1 which defines the Fibonacci function. It is an interesting problem since its derivations are trees, whose dimensions depend on an input argument. The last clause represents a property of the Fibonacci function expressed as an integrity constraint.

We have made a prototype implementation of this approach and performed some experiments on a set of software verification problems, which shows some promise. The main contributions of this paper are as follows.

1. We present a linearisation procedure for dimension-bounded Horn clauses using partial evaluation (Section 3).
2. We give an algorithm for solving a set of non-linear Horn clauses using a linear Horn clause solver (Section 4).
3. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause problems (Section 5).

2 Preliminaries

A constrained Horn clause (CHC) is a first order formula of the form $p(X) \leftarrow C, p_1(X_1), \ldots, p_k(X_k)$ $(k \geq 0)$ (using Constraint Logic Programming syntax), where $C$ is a conjunction of constraints with respect to some constraint theory, $X_1, X$ are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $C, p_1(X_1), \ldots, p_k(X_k)$ is the body. An atomic formula, or simply atom, is a formula $p(t)$ where $p$ is a non-constraint predicate symbol and $t$ a tuple of arguments. Atoms are sometimes written as $A, B$ or $H$, possibly with sub- or superscripts.

A clause is called non-linear if it contains more than one atom in the body, otherwise it is called linear. A set of Horn clauses $P$ is called linear if $P$ only contains linear clauses, otherwise it is called non-linear. Integrity constraints are a special kind of Horn clauses whose head is $false$ where $false$ is always interpreted as $FALSE$. A set of Horn clauses is sometimes called a (constraint logic) program.

An interpretation of a set of CHCs is represented as a set of constrained facts of the form $A \leftarrow C$ where $A$ is an atomic formula $p(Z)$ where $Z$ is a tuple of distinct variables and $C$ is a constraint over $Z$ with respect to some constraint theory. An interpretation that makes each clause in $P$ TRUE is called a model of $P$. We say a set of Horn clause $P$ (including integrity constraints) is safe (solvable) iff it has a model. In some works e.g. [5, 29], a model is also called a solution and we use them interchangeably in this paper.

**Figure 1:** Example CHCs Fib defining a Fibonacci function.

c1. fib(A, B):- A>=0, A=<1, B=A.
c2. fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
c3. false:- A>5, fib(A,B), B<A.
A labeled tree \( c(t_1, \ldots, t_k) \) \((n \geq 0)\) is a tree whose nodes are labeled by identifiers, where \( c \) is the label of the root and \( t_1, \ldots, t_k \) are labeled trees, the children of the root.

**Definition 1 (Tree dimension (adapted from [12]))** Given a labeled tree \( t = c(t_1, \ldots, t_k) \), the tree dimension of \( t \) represented as \( \dim(t) \) is defined as follows:

\[
\dim(t) = \begin{cases} 
0 & \text{if } k = 0 \\
\max_{i \in [1..k]} \dim(t_i) & \text{if there is a unique maximum} \\
\max_{i \in [1..k]} \dim(t_i) + 1 & \text{otherwise}
\end{cases}
\]

Given a set of Horn clauses, we associate with each clause \( p(X) \leftarrow C \in P \), then \( H = 0 \leftarrow C \in P^{\leq k} \). An atom with predicate \( p^{\leq d} \) or \( p = d \) is denoted \( H^{\leq d} \) or \( H = d \) respectively. Such atoms have derivation trees of dimension at most \( d \) and exactly \( d \) respectively.

**Definition 2 (At-most-\( k \)-dimension program \( P^{\leq k} \))** Let \( P \) be a set of CHCs. \( P^{\leq k} \) consists of the following clauses (adapted from [28, 24]):

1. Linear clauses:
   - If \( H \leftarrow \mathcal{C} \in P \), then \( H = 0 \leftarrow \mathcal{C} \in P^{\leq k} \).
   - If \( H \leftarrow \mathcal{C}, B_1 \in P \) then \( H^{=d} \leftarrow \mathcal{C}, B_1^{=d} \in P^{\leq k} \) for \( 0 \leq d \leq k \).

2. Non-linear clauses:
   - If \( H \leftarrow \mathcal{C}, B_1, B_2, \ldots, B_r \in P \) with \( r > 1 \) and one of the following holds:
     - For \( 1 \leq d \leq k \), and \( 1 \leq j \leq r \):
       - Set \( Z_j = B_j^{=d} \) and \( Z_i = B_i^{=d-1} \) for \( 1 \leq i \leq r \) and \( i \neq j \). Then: \( H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leq k} \).
Solving non-linear Horn clauses using a linear Horn clause solver

\[ \text{linear clauses} \]
1. \text{fib(0)}(A,B) :- A\geq 0, A\leq 1, B=A.
2. \text{false(0)} :- A>5, B<A, \text{fib(0)}(A,B).
3. \text{false[0]} :- false(0).
4. \text{fib[0]}(A,B) :- \text{fib}(0)(A,B).

Figure 3: Fib\[0\] : at-most-0-dimension program of Fib.

- For \( 1 \leq d \leq k \), and \( J \subseteq \{1,\ldots,r\} \) with \( |J| = 2 \):
  
  Set \( Z_i = B_i^{d-1} \) if \( i \in J \) and \( Z_i = B_i^{d-2} \) if \( i \in \{1,\ldots,r\} \setminus J \). If all \( Z_i \) are defined, i.e., \( d \geq 2 \) if \( r > 2 \), then: \( H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leq k} \).

\( P^{\leq k} \) is also called the \( k \)-dimension-bounded program corresponding to \( P \). When the value of \( k \) is not important, any program generated using the Definition 2 is called a dimension-bounded program. The relation between \( P \) and its \( k \)-dimensional program is given by in the Proposition 1 where \( \models \) is the usual “logical consequence” operator.

**Proposition 1 (Relation between \( P \) and \( P^{\leq k} \))** Let \( P \) be a program and \( P^{\leq k} \) \((k \geq 0)\) be its \( k \)-dimensional program. Let \( p(t) \) be an atom where \( p \) is a predicate of \( P \) and \( p^*(t) \) \((\ast \in \{=,\leq\})\) be an atom where \( p^* \) is a predicate of \( P^{\leq k} \). Then we have: \( P^{\leq k} \models p^*(t) \implies P \models p(t) \).

In other words, Proposition 1 says that the set of facts that can be derived from \( P^{\leq k} \) is a subset of the set of facts that can be derived from \( P \), taking the predicate renaming into account. In this sense \( P^{\leq k} \) is an under-approximation of \( P \). In particular, if \( P^{\leq k} \models \text{false}^* \) then \( P \models \text{false} \).

Let \( S \) be an interpretation of a dimension-bounded set of clauses \( P^{\leq k} \). That is, \( S \) is a set of constrained facts of the form \( H^{=d} \leftarrow \mathcal{C} \) or \( H^{\leq d} \leftarrow \mathcal{C} \). An interpretation of \( P \) is constructed from \( S \) as follows.

**Definition 3** (\( S^{P^{\leq k}} \): an interpretation of \( P \) constructed from an interpretation of \( P^{\leq k} \)) Let \( S \) be an interpretation of \( P^{\leq k} \). Then \( S^{P^{\leq k}} \) is the following set of constrained facts.

\[
S^{P^{\leq k}} = \{ p(X) \leftarrow \bigvee \{ \mathcal{C} \mid p^{=d}(X) \leftarrow \mathcal{C} \in S \text{ and } p^{\leq d}(X) \leftarrow \mathcal{C} \in S \} \mid d \in \{0..k\} \text{ and } p \text{ is a predicate in } P \}
\]

The set \( S^{P^{\leq k}} \) is a disjunctive interpretation of \( P \) where the interpretation of \( p \) is the disjunction of the interpretations of the corresponding dimension-bounded versions of \( p \) in \( P^{\leq k} \).

The at-most-0-dimension program of Fib in Figure 1 is depicted in Figure 3. In textual form we represent a predicate \( p^{\leq k} \) by \( p[k] \) and a predicate \( p=k \) by \( p(k) \). The at-most-1-dimension program of Fib in Figure 1 is depicted in Figure 4. Note that 0-dimension program is included in 1-dimension program. In general, all the clauses in \( P^{\leq k} \) are also in \( P^{\leq k+1} \). This provides a basis for an iterative strategy for a bounded set of Horn clauses. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. It only becomes a viable approach if a solution of \( p^{\leq k} \) for some \( k \geq 0 \) is general enough to hold for all dimensions of \( P \).
3 Linearisation strategies for dimension-bounded set of Horn clauses

In this section, we present linearisation strategies for set of clauses of bounded dimension. It is known \[1\] that a dimension-bounded set of clauses can be linearised, preserving satisfiability. In this section we describe a practical technique for linearisation, based on partial evaluation of an interpreter.

3.1 Linearisation based on partial evaluation

*Partial evaluation* (PE) has been studied for a variety of languages including logic programs \[21,14,25,22,27\]. We follow the pattern of transforming a program (a set of Horn clauses) by specialising an interpreter for that program \[13,22\]. Let \( PE \) be a partial evaluator, \( I \) an interpreter and \( P \) an object program. Then the partial evaluation of \( I \) with respect to \( P \), denoted \( PE(I,P) \), represents the “compilation” of \( P \) using the semantics given by \( I \).

We first write an interpreter for Horn clause programs, which is also written as a set of Horn clauses. Given a (possibly empty) conjunction of atoms (called a *goal*) the interpreter constructs a derivation, implementing a standard left-to-right, depth-first search. In the interpreter predicate \( \text{solve}(Gs) \), \( Gs \) is the goal, represented as a list of atoms. The basic step of the interpreter is represented by the clauses for \( \text{solve}(Gs) \) shown in Figure 5. If the conjunction is not empty, its first atom \( G \) is selected along with a matching Horn clause \( G \leftarrow Cs,B \) in the program being interpreted, where \( Cs \) is a conjunction of constraints and \( B \) is a conjunction of atoms. This clause is represented by \( \text{hornClause}(G,Cs,B) \) in the interpreter. The body of the clause is conjoined with the remaining goal atoms, and the derivation continues with the new goal \( Gs1 \). If the conjunction is empty, the derivation is successful (second clause).

To interpret a dimension-bounded set of clauses (say the bound is \( k \)), we use the fact that in all successful runs of the interpreter in which goals are selected in increasing order of dimension, the size of the conjunction of goals (that is, the length of the argument of \( \text{solve} \)) has an upper bound related to \( k \). This bound is known as the *index* of the set of clauses and is given as \((i-1) \ast k + 1\), where \( i \) is the maximum number of non-constraint atoms in the body of clauses \[11\]. Given this index, we can augment the interpreter with a check on the size of the conjunction, ensuring that it never exceeds the
solve([G|Gs]) :-
    hornClause(G,Cs,B), solveConstraints(Cs), append(B,Gs,Gs1),
    solve(Gs1).
solve([]).

Figure 5: Depth-first interpreter for Horn clauses

index. In addition, due to the requirement of increasing dimension in the selection of atoms, a left-to-right computation rule is not sufficient; therefore we permute the set of atoms in each clause body, since in at least one permutation the goals will be ordered by dimension. With these changes the interpreter remains complete for clauses of the given maximum index, at the possible cost of some redundancy in the search.

These additions result in the interpreter whose top level is shown in Figure 6. Let the interpreter predicate solve(Gs,Index,L) mean that the conjunction of goals Gs is to be solved, and L, Index are numbers representing the size of Gs and the maximum size of the stack of goals.

\begin{verbatim}
go(Index) :-
    solve([false],Index,1).
solve([G|Gs],Index,L) :-
    hornClause(G,Cs,B), solveConstraints(Cs),
    length(B,L1), L2 is L1+L-1, L2 =< Index,
    perm(B,B1), append(B1,Gs,Gs1),
    solve(Gs1,Index,L2).
solve([],_,_).
\end{verbatim}

Figure 6: Interpreter for linearisation

**Partial evaluation of the interpreter.** Given a set of facts of the form hornClause(G,Cs,B) representing the Horn clauses to be linearised, and some value of the index K, the interpreter can be partially evaluated. We use Logen \cite{Logen} to perform the partial evaluation with respect to a call to go(Index), which initiates a proof of the goal false (see first clause of interpreter). All interpreter computations are partially evaluated except for the calls to solve(Gs,Index,L) and the execution of constraints within the goal solveConstraints(Cs). Furthermore Logen performs standard structure-flattening and predicate renaming operations, yielding a set of clauses of the form solve_i(X) :- Cs, solve_j(Y), where solve_i(X) and solve_j(Y) are instantiations of solve(Gs,Index,L) and Cs is a constraint. Thus the resulting clauses are linear, and furthermore preserve the meaning of the original clauses as given by the interpreter, by correctness of the partial evaluation procedure. The linearisation procedure is independent of the constraint theory underlying the clauses.

**Proposition 2** Let P be a program and P \( \leq k \) \((k \geq 0)\) be its k-dimensional program. Let i be the maximum number of atoms in clause bodies of P \( \leq k \). Let Index = \( (i - 1) * k + 1 \). Let P' be a partial evaluation of the interpreter in Figure 5 with respect to P and the goal go(Index). Then P \( \leq k \) \( \models \) false \( \leq k \) iff P' \( \models \) go(Index).

Furthermore P' is linear if the partial evaluator follows the strategy described above. Combining Propositions 2 and 1, we conclude that P' \( \models \) go(Index) \( \Rightarrow \) P \( \models \) false.
Note that linearisation required partial evaluation of the perm predicate, giving a blow-up in program size related to the length of the clause bodies. This is further discussed at the end of Section 5.

3.2 Obtaining linear over-approximations with a partial model

First we note that the set of predicates in \( P_{\leq k} \) is a subset of the set of predicates in \( P_{\leq k+1} \). Given a model \( M \) for the predicates in \( P_{\leq k} \), \( P_{\leq k+1} \) can be linearised if we replace each occurrence of a predicate from \( P_{\leq k} \) in the body of a clause in \( P_{\leq k+1} \) with the corresponding constraint from the model \( M \). The resulting set of clauses is linear since \( P_{\leq k+1} \) contains at most one predicate in its body from \( P_{\leq k+1} \) which is not in \( P_{\leq k} \). Furthermore if \( P_{\leq k+1} \) has a model then so does the set of clauses resulting from the replacement; the converse is however not the case since the model \( M \) represents an over-approximation of \( P_{\leq k} \).

More generally, we can replace any subset of the occurrences of predicates from \( P_{\leq k} \) in \( P_{\leq k+1} \). We summarise this in the following lemma.

**Lemma 1 (Linear over-approximation)** Let \( M \) be a model of the predicates in \( P_{\leq k} \), represented by a set of “constrained facts” \( p(X) \leftarrow \mathcal{C} \) where \( p \) is a predicate in \( P_{\leq k} \) and \( \mathcal{C} \) is the constraint satisfying \( p \) in \( M \). Let \( P' \) be any set of clauses obtained from \( P_{\leq k+1} \) by replacing some of the occurrences of predicates \( p(X) \) from \( P_{\leq k} \) in the bodies of clauses in \( P_{\leq k+1} \) with their corresponding interpretation \( \mathcal{C} \) in \( M \). Then

1. If \( P_{\leq k+1} \) has a model then so does \( P' \);
2. If \( P' \) contains no predicate from \( P_{\leq k} \), then \( P' \) is linear.

4 Algorithm for solving sets of non-linear Horn clauses

A basic procedure for solving a set of non-linear Horn clauses using a linear Horn clause solver is presented in Algorithms 1 and 2. From now on we use linear solver for linear Horn clause solver for simplicity. The main procedure \( \text{SOLVE}(P) \) takes a set of non-linear Horn clauses \( P \) as input and outputs upon termination \( \text{(safe, solution)} \) if \( P \) is solvable or \( \text{(unsafe, counterexample)} \) otherwise. We represent counterexample as a trace tree. For a linear program it corresponds to a sequence of clauses.

**Definition 4** \((S_t)\) Let \( S \) be an interpretation of a set of Horn clauses \( P \). Let \( t \) be any trace tree of \( P \) and \( T_t \) be its corresponding Horn clause derivation. Then we denote by \( A_t \) the set of non-constraint atom label of \( T_t \). Then \( S_t \) is defined to be the set

\[
S_t = \{(H \leftarrow \mathcal{C}) \mid (H \leftarrow \mathcal{C}) \in S \land H \not\in A_t\}.
\]

Algorithm 2 is an extended version of Algorithm 1 which uses the solution for \( P_{\leq k} \) to help to linearise \( P_{\leq k+1} \) and also allows a more refined termination condition based on whether or not the solution for \( P_{\leq k} \) is used in constructing a counterexample for \( P_{\leq k+1} \).

The procedures make use of several sub-procedures which will be described next.

4.1 Components of the algorithm

- **KDIM(P,k)**: produces an at-most-\( k \)-dimension program \( P_{\leq k} \) (Definition 2). By definition, \( P_{\leq k} \) is linear for \( k = 0 \). For our example program presented in Figure 1, \( \text{Fib}_{\leq 0} \) is shown in Figure 3, which is linear since there is at-most one non-constraint atom in the body of each clauses.
Solving non-linear Horn clauses using a linear Horn clause solver

- **SOLVE LINEAR**($P'$): solves a set of linear Horn clauses $P'$. We assume the following about a linear solver: (i) if it terminates on $P'$, then it returns either safe and a solution or unsafe and a counterexample; (ii) it is sound, that is, if it returns a solution $S$ for $P'$ then $P'$ has a model and $S$ is a solution (model) of $P'$; if it returns unsafe and a counterexample $cEx$ then $P'$ is unsafe and $cEx$ is a witness. In our setting (Algorithms 1 and 2), $P'$ corresponds to a linearised version of $P^{\leq k}$ for some $P$ and $k \geq 0$. For technical reasons, the top level predicate false[$k$] of $P^{\leq k}$ if any, is renamed to false before passing to a linear solver.

In essence, any Horn clause solver which complies with our assumption, for example QARMC [17], Convex polyhedral analyser [23], ELDARICA [20] etc. can be used in a black-box fashion but in this paper, we make use of a solver described in [23], which is based on abstract interpretation [6] over the domain of convex polyhedra [7] but without refinement using finite tree automata.

The solver produces the following solution for the program in Figure 3. We can check it is in fact a solution (model).

```prolog
fib(0)(A,B) :- [-A>= -1,A>=0,B=A].
fib[0](A,B) :- [-A>= -1,A>=0,B=A].
false[0] :- <>; % <> means that there is no model for false[0],
% so we can discard it
```

- **LINEARISE**($P,k,S$) generates a linear set of clauses from $P^{\leq k}$ and an interpretation $S$ for $P^{\leq k}$. Let $S$ be a set of constrained facts of the form $p(X) \leftarrow \mathcal{C}$, where $p$ is a predicate from $P^{\leq k}$, the procedure replaces every clause from $P^{\leq k}$ with head $p(X)$ by $p(X) \leftarrow \mathcal{C}$. This produces a set of clauses say $P'$. Then the procedure **LINEARISE PE**($P',M$) is called, which is the linearisation procedure based on partial evaluation described in Section 3 where $M$ is a bound for the stack usage for linearising $P^{\leq k}$.

An excerpt from Fib$^{\leq 1}$ is shown below.

```prolog
false(1) :- A>5, B<A, fib(1)(A,B).
fib(1)(A,B) :- A>1, C=A-2, E=A-1, B=F+D, fib(1)(C,D), fib[0](E,F).
fib(0)(A,B) :- B=A, A=<1, A>=0.
```

After reusing the solution obtained for Fib$^{\leq 0}$ and linearising, we obtain the following set of linear clauses.

```prolog
false(1) :- A>5, B<A, fib(1)(A,B).
```

Continuing to run our algorithm, the following solution obtained for Fib$^{\leq 2}$ becomes a solution for the program in Figure 1 (the original program) and the algorithm terminates.

```prolog
fib(0)(A,B) :- [-A>= -1,A>=0,B=1].
fib[0](A,B) :- [-A>= -1,A>=0,B=1].
fib(1)(A,B) :- [A>=2,A+ -B=0].
fib[1](A,B) :- [A+ -B>= -1,B>=1,-A+B>=0].
fib(2)(A,B) :- [A>=4,-2*A+B>= -3].
fib[2](A,B) :- [A>=0,B>=1,-A+B>=0].
```
Algorithm 1: Algorithm for solving a set of Horn clauses

1. Procedure `SOLVE(P)`
   - **Input**: A set of CHCs $P$
   - **Output**: `(safe, solution), (unsafe, cex)`
2. $k \leftarrow 0$
3. $P' \leftarrow \text{LINEARISE}(P, k, 0)$;
4. `(status, Result) \leftarrow \text{SOLVE_LINEAR}(P')`; /* Result is a solution or a cex */
5. if `status = safe` then
6.   if $(\text{Result}^P)$ is a solution of $P$ then return $(\text{safe}, \text{Result}^P)$;
7.   $k \leftarrow k + 1$;
8. else
9.   return $(\text{unsafe}, \text{Result})$; /* Result is a cex */
10. goto

Algorithm 2: Algorithm for solving a set of Horn clauses, with reuse of lower dimension solutions

1. Procedure `SOLVE(P)`
   - **Input**: A set of CHCs $P$
   - **Output**: `(safe, solution), (unsafe, cex)`
2. $k \leftarrow 0$
3. $S \leftarrow \emptyset$
4. $P' \leftarrow \text{LINEARISE}(P, k, S)$;
5. `(status, Result) \leftarrow \text{SOLVE_LINEAR}(P')`; /* Result is a solution or a cex */
6. if `status = safe` then
7.   if $(\text{Result}^P)$ is a solution of $P$ then return $(\text{safe}, \text{Result}^P)$;
8.   $k \leftarrow k + 1$;
9.   $S \leftarrow \text{Result}$;
10. else
11.   if $S \cap \text{Result}_{|cEx} = \emptyset$ then return $(\text{unsafe}, \text{Result})$; /* Result is a linear cex */
12.   $S \leftarrow S \setminus \text{Result}_{|cEx}$; /* Result$_{|cEx}$: Definition 4 */
13. goto

Algorithm 3: Algorithm for linearising a set of clauses

1. Procedure `LINEARISE(P, k, S)`
   - **Input**: A set of CHCs $P$, an integer $k$ and a set of constrained facts $S$
   - **Output**: A linearised set of clauses $P_{lin}$
2. $P^{\leq k} \leftarrow \text{KDIM}(P, k)$; /* Definition 2 */
3. $P' \leftarrow \text{SUBSTITUTE}(P^{\leq k}, S)$; /* substitute atoms of $P^{\leq k}$ with their interpretations from $S$ */
4. $M \leftarrow (i - 1) \ast k + 1$; /* where $i$ is the maximal number of body atoms of $P$ */
5. $P_{lin} \leftarrow \text{LINEARISE_PE}(P', M)$; /* Section 3.1 */
6. return $P_{lin}$
4.2 Reuse of solutions, refinement and linearisation

Algorithm 2 solves non-linear Horn clauses \( P \) in essentially the same way as Algorithm 1 but incorporates a refinement phase in the case that the linear solver finds a counterexample. This counterexample possibly uses some of the model of the lower-dimension predicates \( S \), in which case it is not certain whether it is a false alarm or a real counterexample. If the counterexample did use some of the predicate solutions from \( S \), then we discard those solutions (Algorithm 2, line 13) and return to the linearisation step. If the counterexample does not use any predicate solutions from \( S \), then it is a real counterexample (Algorithm 2, line 12). We will clarify this with an example program (linear for simplicity) shown below.

\begin{align*}
\text{c1. } \text{false} & : \text{X}=0, p(X). \\
\text{c2. } \text{false} & : q(X). \\
\text{c3. } p(X) & : \text{X}>0. \\
\text{c4. } q(X) & : \text{X}=0.
\end{align*}

Suppose we have an approximate solution \( S = \{ p(X) : \text{true} \} \) for the predicate \( p(X) \). Using this solution, the above program is transformed into the following program.

\begin{align*}
\text{c1. } \text{false} & : \text{X}=0, p(X). \\
\text{c2. } \text{false} & : q(X). \\
\text{c3. } p(X) & : \text{true}. \text{(approximate solution)} \\
\text{c4. } q(X) & : \text{X}=0.
\end{align*}

The trace \( c_1(c_3) \) is a counterexample for this transformed program but not to the original program (since it uses approximate solution for the predicate \( p \)). However the trace \( c_2(c_4) \) is a counterexample for this program as well as to the original since it does not use any approximate solution for the predicates appearing in the counterexample.

A schematic overview of Algorithm 2 is shown in Figure 7. At each iteration of the abstraction-refinement loop, the at-most-\( k \)-dimension under-approximation of \( P \) is computed, then linearised and solved using a solver for linear Horn clauses.

![Abstraction-refinement scheme](image)

**Figure 7:** Abstraction-refinement scheme for solving non-linear Horn clauses using a solver for linear Horn clauses. \( P' \) is a set of linear CHC obtained by linearising the at-most-\( k \)-dimension under-approximation, \( \leq k \), of \( P \).

The soundness of Algorithms 1 and 2 is captured by Proposition 3.
Proposition 3 (Soundness) If Algorithm 1 or 2 returns safe and a solution $S$ for a set of clauses $P$ then $P$ is safe and $S$ is in fact a solution of $P$; if it returns unsafe and a counterexample $cEx$ then $P$ is unsafe and $cEx$ is a witness.

Another desired property of the Algorithm 2 is that of progress, that is, the same counterexample does not arise more than once.

5 Experimental results

We made a prototype implementation of Algorithm 2 in the tool called LHornSolver. It uses the solver described in [23] without refinement as a linear Horn clause solver. LHornSolver is written in Ciao prolog [19] and is interfaced with Parma Polyhedra Library [2] and Yices SMT solver [10] for handling constraints. Then we carried out an experiment on a set of 44 non-linear CHC verification problems taken from the repository of software verification benchmarks, the recursive category of SV-COMP [4] and the tool QARMC. The experiments were run on a MAC computer running OS X on 2.3 GHz Intel core i7 processor and 8 GB memory. The benchmarks that we use in the experiments are not beyond the capabilities of existing solvers, but they are challenging. These programs are first translated to Prolog syntax using the tools Eldarica [20] and SeaHorn [18]. Our aim with these experiments is to explore: (1) whether using a linear solver for non-linear problem solving is practical; (2) the relationship between the solvability of a problem and its dimension; and (3) how the current results compare with the results using the state of the art non-linear Horn clause verification tool (in our case RAHFT [23]). The results are summarized in Table 1.

In the table Program represents a program, Safety represents a verification result, # Itr. and Time (s) successively represent the number of refinement iterations and the time in seconds need to solve a program using both RAHFT and LHornSolver. It is to note that the underlying abstract interpreter, that is, the convex polyhedral analyser (CPA) is the same for both RAHFT and LHornSolver but LHornSolver uses it to solve linear Horn clauses though the CPA is not optimised for linear problems. The column # Itr. for LHornSolver represents a value of $k$ for which a solution of $P \leq k$ (under-approximation) of a set of clauses $P$ becomes a solution for $P$ or $P \leq k$ becomes unsafe. The symbol “?” means that the result is unknown within the given time bound. The result “safe” means that the program is safe (solvable) and “unsafe” means it is unsafe.

LHornSolver solves 27 out of 44 (about 61%) problems within a second. In most of these problems, a solution of an under approximation ($P \leq k$) becomes a solution for the original program or $P \leq k$ becomes unsafe for a fairly small value of $k$ (1 or 2). This suggests that the solvability of a problem is shallow with respect to its dimension. This demonstrates the feasibility of solving a set of non-linear Horn clauses using a solver for linear Horn clauses.

In contrast, RAHFT solves all the problem. The difference in results maybe due to the following reason: the linear solver that is used in LHornSolver is the CPA (without refinement in contrast to [23]). The solver terminates but produces false alarms. If we use CPA with refinement as in [23], then we lose predicates names (due to program transformation), so the solution or counterexamples produced by the tool do not correspond to the original program (it is very hard to keep track of the changes). This hinders the reuse of solution from lower dimension to linearise program of higher dimension or refine it.

https://github.com/bishoksan/LHornSolver
https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/
http://sv-comp.sosy-lab.org/2015/benchmarks.php
https://github.com/uuverifiers/eldarica
<table>
<thead>
<tr>
<th>Program</th>
<th>RAHFT</th>
<th>LHornSolver</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Safety</td>
<td># iter.</td>
</tr>
<tr>
<td>Addition03_false-unreach</td>
<td>safe</td>
<td>2</td>
</tr>
<tr>
<td>McCarthy91_false-unreach</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>addition.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>bfprt.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>binarysearch.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>countZero.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>eq.horn</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>fib.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>identity.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>merge.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>palindrome.nts.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>parity.nts.pl</td>
<td>unsafe</td>
<td>1</td>
</tr>
<tr>
<td>remainder.nts.pl</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>revlen.pl</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>running.nts.pl</td>
<td>unsafe</td>
<td>1</td>
</tr>
<tr>
<td>sum_10x0_false-unreach</td>
<td>unsafe</td>
<td>10</td>
</tr>
<tr>
<td>sum_non_eq_false-unreach</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>suma1.horn</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>suma2.horn</td>
<td>unsafe</td>
<td>0</td>
</tr>
<tr>
<td>summ_SG1.r.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_SG2.r.horn</td>
<td>safe</td>
<td>8</td>
</tr>
<tr>
<td>summ_SG3.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_b.horn</td>
<td>safe</td>
<td>2</td>
</tr>
<tr>
<td>summ_binsearch.horn</td>
<td>safe</td>
<td>1</td>
</tr>
<tr>
<td>summ_cil.casts.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_formals.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_g.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_globals.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_h.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_local-ctx-call.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summlocals.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summlocals2.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summlocals3.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summlocals4.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_mccarthy2.horn</td>
<td>safe</td>
<td>3</td>
</tr>
<tr>
<td>summ_multi-call.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_nested.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_ptr_assign.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_recursive.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_rholocal.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_rholocal2.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_slicing.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_summs.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_typedef.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td>summ_x.horn</td>
<td>safe</td>
<td>0</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>0.64</td>
<td>2.3</td>
</tr>
</tbody>
</table>

Table 1: Experimental results on non-linear CHC verification problems with a timeout of 5 minutes.
using the counterexample trace. Other solvers which don’t modify the programs but produce solutions or counterexamples can be used as a linear solver in principle and we leave it for the future work. Another disadvantage of using CPA is that, if it cannot solve a linear program, then it emits an abstract trace which is checked for a feasibility. If it is spurious then LHornSolver returns with unknown (in principle we can refine the program but the refinement will have the problem as mentioned above). So it is highly unlikely that the trace picked by the tool non-deterministically results to be a real counterexample. We noticed in our experiments that the trace picked was spurious most of the times and LHornSolver immediately returned “unknown” answer before the timeout. This also explains why solving time of LHornSolver is less than that of RAHFT.

The interpreter described in Figure 6 computed a permutation of the atoms in a clause body; partial evaluation of the permutation procedure can cause a blow-up of the size of the linearised program, relative to the number of atoms in clause bodies. In our experiment because the maximum number of atoms in the bodies of the clauses in our benchmark programs was 5 and the value of $k$ was relatively small ($k = 0 \ldots 2$). The permutation procedure can be avoided if we first generate an at-most-$k$-dimension program whose body atoms are ordered by increasing dimension. This needs unfolding of the $\varepsilon$-clauses, since atoms whose predicate is $p^\varepsilon_d$ cannot be ordered directly; only atoms with predicates of the form $p^{=d}$ can be ordered. We have not yet evaluated the trade-offs in these two approaches.

6 Related Work

In the world of Horn clause solvers, after fixing a constraint theory, we can distinguish solvers depending on whether they can handle general non-linear Horn clauses or not. A majority of solvers [18, 16, 30, 29, 23] handle non-linear Horn clauses but there are notable exceptions like VeriMAP [8] or Sally [5]. For both VeriMAP and Sally, their underlying reasoning engine handles only linear Horn clauses which restricts, in principle, their applicability. Our contribution is to lift this restriction by allowing those tools to be applied on arbitrary sets of Horn Clauses, linear or not, through a linearisation procedure that underapproximates the set of solutions. We give empirical evidence that this underapproximation often provides enough coverage to enable the verification of the original set of Horn clauses. To summarize, we allow solvers with restrictions to be applied on any input at the price of an under-approximation which often results in full coverage.

Our linearisation method based on partial evaluation described in Section 3.1 is related to the linearisation method based on fold-unfold transformations described by De Angelis et al. [9]. While their procedure transforms the target set of clauses directly, we transform an interpreter for the clauses using a generic partial evaluation procedure. Any clause transformation procedure could be formulated as a meta-program and partial evaluation applied to that program to yield the specified transformation. Thus neither approach offers any more power than the other. However the use of partial evaluation is arguably more flexible. The interpreter that is partially evaluated in our procedure is a standard interpreter for Horn clauses, modified with a bound on the size of goals, directly incorporating a general result that there is an upper bound on the size of goals in derivations with dimension-bounded programs. This provides a very generic starting point for the transformation with an explicit relation to the semantics of the clauses. A whole family of similar transformations could be formulated by varying the interpreter (for example using breadth-first search). The procedure in [9] is tailored to a restrictive setting where only goal clauses (integrity constraints) are non-linear and rest of the clauses are linear; correctness has to be established for that case.

[https://github.com/SRI-CSL/sally]
Ganty, Josif and Konečný [15] used the notion of tree dimension for computing summaries of procedural programs by underapproximating them. Roughly speaking, they compute procedure summaries iteratively, starting from the program behaviors captured by derivation trees of dimension 0. Then they reuse these summaries to compute summaries for program behaviors captured by derivation trees of dimension 1 and so on for 2, 3, etc. Kafle, Gallagher and Ganty [24] adapted the idea of dimension-based underapproximations to the setting of Horn clause systems. They gave empirical evidence supporting the thesis that for small values of the dimension the solutions are general enough to hold for every dimension. Their approach still required the use of general Horn-clause solvers capable of handling non-linear clauses. In this paper, we lift this requirement and allow the use of solvers for linear clauses only. Moreover, we provide an abstraction refinement loop that enables the solutions for lower dimension to be reused when searching for solutions in higher dimension.

7 Conclusion and future work

We presented an abstraction-refinement approach for solving a set of non-linear Horn clauses using an off-the-shelf linear Horn clause solver. It was achieved through a linearisation of a dimension bounded set of Horn clauses (which are known to be linearisable) using partial evaluation and the use of a linear Horn clause solver. Experiment on a set of non-linear Horn clause verification problems using our approach shows that the approach is feasible (a linear solver can be used for solving non-linear problems) and the solvability of a problem is shallow with respect to its dimension.

In the future, we plan to compare our results with the results from a specialised linear Horn clause solver like VeriMap and other non-linear Horn clause solvers. We also plan to experiment with different linearisation strategies for Horn clauses and study their effects in Horn clause verification.

Acknowledgement

The authors would like to thank José F. Morales for his help with Ciao Prolog foreign language interface and some parts of the implementation.

References


