Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 00 (2015) 1–19

Procedia Computer Science

Horn clause verification with convex polyhedral abstraction and tree automata-based refinement

Bishoksan Kafle*

Roskilde University, Denmark

John P. Gallagher*

Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain

Abstract

In this paper we apply tree-automata techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions; firstly we handle tree automata rather than string automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision. We show how to derive a refined set of Horn clauses in which given infeasible traces have been eliminated, using a recent optimised algorithm for tree automata determinisation. We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

Keywords:
Horn clauses, Abstract interpretation, Finite tree automata, Tree automata determinisation

1. Introduction

The formalism of Constrained Horn clauses (CHCs), as an intermediate language for verification of programs in various languages, has become popular due to its well understood properties and expressiveness; this has led to a range of tools for analysis and verification of CHCs. Given a program and a property ϕ to be verified, a set of CHCs V, such that V is satisfiable if and only if ϕ holds, is called a *verification condition* for ϕ. CHC verification conditions can be obtained from imperative, functional or concurrent languages, among others, by a variety of semantics-based techniques including big- and small-step semantics, Hoare triples, or other intermediate forms such as control-flow graphs [1, 2, 3, 4, 5, 6]. We do not consider the process of generating CHC verification conditions in this paper.

There are several approaches to checking the satisfiability of CHC verification conditions, including abstract interpretation and counterexample-guided abstraction refinement (see Section 7). In this paper we apply tree-automata

*Corresponding author
Email addresses: kafle@ruc.dk (Bishoksan Kafle), jpg@ruc.dk (John P. Gallagher)

techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions [7]; firstly, we handle tree automata rather than word automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision.

Our approach is similar in spirit to counterexample-guided abstraction refinement (CEGAR) or iterative specialisation approaches, in which a refined set of clauses is generated by eliminating one or more of the infeasible paths from the original set of clauses until the safety or unsafety of the clauses is proven. More specifically, we show how to construct tree automata capturing both the traces (derivations) of a given set of Horn clauses and also one or more infeasible traces discovered after abstract interpretation of the clauses. From these we construct a refined automaton in which the infeasible trace(s) have been eliminated and a new set of clauses is constructed from the refined automaton. This guarantees progress in that the same infeasible trace cannot be generated (in *any* abstract interpretation). In addition, the clauses are restructured during the elimination of the trace, which can lead to more precise abstractions in subsequent iterations. The refinement is manifested in the refined clauses, rather than in an accumulated set of properties as in the CEGAR [8] approach. We rely on the abstract interpretation of the clauses to generate useful properties, rather than hoping to find them during the refinement itself.

We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. This splitting induces splitting in the predicates of the original set of clauses and its analysis using convex polyhedra leads to disjunctive abstractions. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

The main contributions of this paper are the following; (1) We construct a correspondence between computations using Horn clauses and finite tree automata (FTA) (Section 4). (2) We construct a refined set of clauses directly from a tree automaton representation of the clauses and an infeasible trace; the trace is eliminated from the refined clauses (Section 4.4). (3) We propose a "splitting" operator on FTAs (Section 3) and describe its role in Horn clause verification (Section 5.1). (4) We demonstrate the feasibility of our approach in practice applying it to Horn clause verification problems (Section 6).

This paper is an extended version of [9]. The paper has been extended in the following directions: (1) The proofs of all propositions have been provided; (2) further information about the implementation of our tool chain is given (see Section 6); (3) further experiments comparing our results with the state of the art verification tools in the literature are provided (see Sub-sections 4.5.1 and 4.5.2); (4) further experimental results on some additional bechmarks from the software verification competition 2015 have been provided (see Sub-section 6.5).

2. Summary of our approach

To motivate readers, we present an example set of CHCs P in Figure 1 which will be used throughout this paper. This is an interesting problem in which the computations are trees rather than linear sequences.

```
c1. mc91(A,B) :- A > 100, B = A-10.
c2. mc91(A,B) :- A =< 100, C = A+11, mc91(C,D), mc91(D,B).
c3. false :- A =< 100,  B > 91, mc91(A,B).
c4. false :- A =< 100, B =< 90, mc91(A,B).
```

Figure 1. Example CHCs. The McCarthy 91-function

After applying abstract interpretation to this set of clauses, we obtain the following set of constained facts (also called approximation):

```
mc91(A,B) :- [B>90, B>=A-10].
false :- [].
```

Since false is in our approximation, our tool generates an abstract derivation for false which in our case is the clause `c3` followed by the clause `c1` and is represented by a trace term `c3(c1)`. Since this abstract counterexample is

infeasible, our refinement procedure removes this from the set of clauses in Figure 1 to produce a new set of clauses as shown in Figure 2. Our refinement can be viewed as a program transformation guided by a counterexample. From the set of refined clauses, it can be seen that the counterexample `c3(c1)` is impossible to construct. This refinement split the predicate *mc91* of the original clauses and as a result of this we gain some precision. We again analyse the refined clauses using abstract interpretation until its safety or unsafety is proven. In the sections to follow we describe our abstraction-refinement procedure which led to this result in details.

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
```

Figure 2. Refined set of CHCs

The architecture of our abstraction-refinement scheme is shown in the Figure 3. It is accompanied by our main algorithm 1 to give the early picture of our approach.

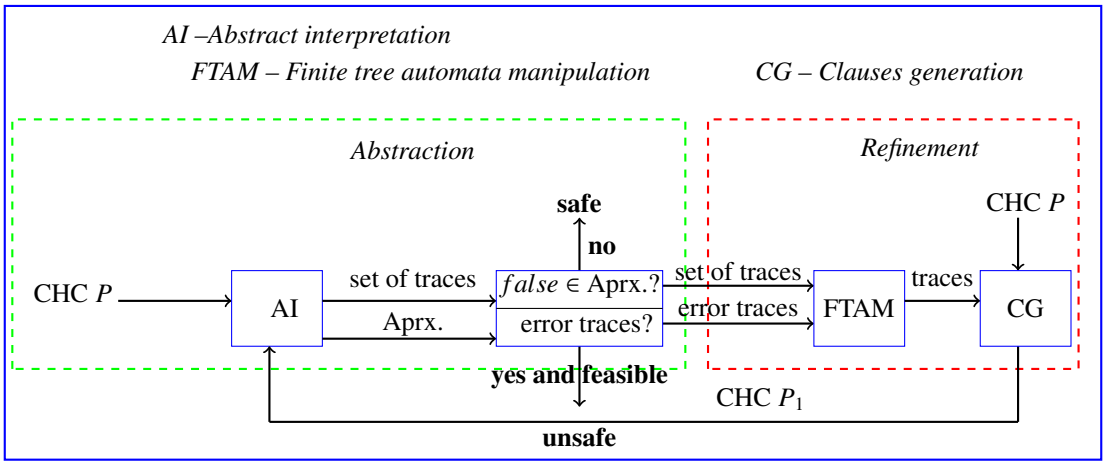

Figure 3. *Abstraction-refinement scheme in Horn clause verification. Aprx. is an approximation produced as a result of abstract interpretation.*

Input: A set of Horn clauses P
Output: *safe* or *unsafe*

1. analyse P using abstract interpretation producing constrained facts M (Algorithm 3);
2. if $\mathsf{false} \notin M$ then **return** *safe* ;
3. if $\mathsf{false} \in M$ then produce derivation t of false using P ;
4. if t is feasible **return** *unsafe* ;
5. $P' \leftarrow \mathit{refinedCls}(P, t)$ (Algorithm 4) ;
9. $P \leftarrow P'$ and goto step 1 ;

Algorithm 1: ALGORITHM for abstraction-refinement of Horn clauses

3. Finite tree automata

Finite tree automata (FTAs) are mathematical machines that define so-called recognisable tree languages, which are possibly infinite sets of terms that have desirable properties such as closure under Boolean set operations and decidability of membership and emptiness.

Definition 1 (Finite tree automaton)**.** *An FTA $\mathcal{A}$ is a tuple (Q, Q_f, Σ, Δ), where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, Σ is a set of function symbols, and Δ is a set of transitions. We assume that Q and Σ are disjoint.*

Each function symbol $f \in \Sigma$ has an arity $n \geq 0$, written as $\mathsf{ar}(f) = n$. The function symbols with arity 0 are called constants. $\mathsf{Term}(\Sigma)$ is the set of ground terms or trees constructed from Σ where $t \in \mathsf{Term}(\Sigma)$ iff $t \in \Sigma$ is a constant or $t = f(t_1, t_2, ..., t_n)$ where $\mathsf{ar}(f) = n$ and $t_1, t_2, ..., t_n \in \mathsf{Term}(\Sigma)$. Similarly $\mathsf{Term}(\Sigma \cup Q)$ is the set of terms/trees constructed from Σ and Q, treating the elements of Q as constants.

Each transition in Δ is of the form $f(q_1, q_2, ..., q_n) \to q$ where $\mathsf{ar}(f) = n$. Given $\delta \in \Delta$ we refer to its left- and right-hand-sides as $\mathsf{lhs}(\delta)$ and $\mathsf{rhs}(\delta)$ respectively. Let $\Rightarrow$ be a one-step rewrite in which $t_1 \Rightarrow t_2$ iff t_2 is the result of replacing one subterm of t_1 equal to $\mathsf{lhs}(\delta)$ by $\mathsf{rhs}(\delta)$, from some $\delta \in \Delta$. The reflexive, transitive closure of $\Rightarrow$ is $\Rightarrow^*$. We say there is a run (resp. successful run) for $t \in \mathsf{Term}(\Sigma)$ if $t \Rightarrow^* q$ where $q \in Q$ (resp. $q \in Q_f$), and we say that t is *accepted* if t has a successful run. An FTA $\mathcal{A}$ defines a set of terms, that is, a tree language, denoted by $\mathcal{L}(\mathcal{A})$, as the set of all terms accepted by $\mathcal{A}$.

Definition 2 (Deterministic FTA (DFTA))**.** *An FTA (Q, Q_f, Σ, Δ) is called bottom-up deterministic iff Δ has no two transitions with the same left hand side.*

We omit the adjective "bottom-up" in this paper and just refer to deterministic FTAs. Runs of a DFTA are deterministic in the sense that for every $t \in \mathsf{Term}(\Sigma)$ there is at most one $q \in Q$ such that $t \Rightarrow^* q$.

3.1. Operations on FTAs

FTAs are closed under Boolean set operations, but for our purposes we mention only union and difference of language of automata, where in addition we assume that the signature Σ is fixed and that the states of FTAs are disjoint from each other when applying operations (the states can be renamed apart).

Definition 3 (Union of FTAs)**.** *Let $\mathcal{A}^1, \mathcal{A}^2$ be FTAs $(Q^1, Q^1_f, \Sigma, \Delta^1)$ and $(Q^2, Q^2_f, \Sigma, \Delta^2)$ respectively. Then $\mathcal{A}^1 \cup \mathcal{A}^2 = (Q^1 \cup Q^2, Q^1_f \cup Q^2_f, \Sigma, \Delta^1 \cup \Delta^2)$, and we have $\mathcal{L}(\mathcal{A}^1 \cup \mathcal{A}^2) = \mathcal{L}(\mathcal{A}^1) \cup \mathcal{L}(\mathcal{A}^2)$.*

Determinisation plays a key role in the theory of FTAs. As far as expressiveness is concerned, we can limit our attention to DFTAs since for every FTA $\mathcal{A}$ there exists a DFTA $\mathcal{A}^d$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^d)$ [10]. The standard construction builds a DFTA $\mathcal{A}^d$ whose states are elements of the powerset of the states of $\mathcal{A}$. The textbook procedure for constructing $\mathcal{A}^d$ from $\mathcal{A}$ [10] is not viewed as a practical procedure for manipulating tree automata, even fairly small ones. In a recent work Gallagher *et al.* [11] developed an optimised algorithm for determinisation, whose worst-case complexity remains unchanged, but which performs dramatically better than existing algorithms in practice. A critical aspect of the algorithm is that the transitions of the determinised automaton are generated in a potentially very compact form called *product form*, which can often be used directly when manipulating the determinised automaton.

Definition 4 (Product Transition)**.** *A* product transition *is of the form $f(Q_1, \ldots, Q_n) \to q$ where Q_i are sets of states and q is a state. The product transition represents a set of transitions $\{f(q_1, \ldots, q_n) \to q \mid q_i \in Q_i, i = 1..n\}$. Thus $\Pi_{i=1}^{n} |Q_i|$ transitions are represented by a single product transition.*

Alternatively, we can regard a product transition as introducing ϵ-transitions. An ϵ-transition has the form $q_1 \to q_2$ where q_1, q_2 are states. ϵ-transitions can be eliminated, if desired. Given a product transition $f(Q_1, \ldots, Q_n) \to q$, introduce n new non-final states $s_1, \ldots, s_n$ corresponding to $Q_1, \ldots, Q_n$ respectively and replace the product transition by the set of transitions $\{f(s_1, \ldots, s_n) \to q\} \cup \{q' \to s_i \mid q' \in Q_i, 1 = 1..n\}$. It can be shown that this transformation preserves the language of the FTA.

Given FTAs $\mathcal{A}^1$ and $\mathcal{A}^2$ there exists an FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$ such that $\mathcal{L}(\mathcal{A}^1 \setminus \mathcal{A}^2) = \mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}^2)$. To construct the difference FTA we use union and determinisation and exploit the following property of determinised states [11].

Property 1. *Let $\mathcal{A}^d$ be the DFTA constructed from $\mathcal{A}$. Let Q be the states of $\mathcal{A}$. Then there is a run $t \Rightarrow^* q$ in $\mathcal{A}$ if and only if there is a run $t \Rightarrow^* Q'$ in $\mathcal{A}^d$ where $Q' \in 2^Q$, such that $q \in Q'$.*

Furthermore recall that a term is accepted by at most one state in a DFTA. This gives rise to the following construction of the difference FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$. We first form the DFTA for the union of the two FTAs and then remove those of its final states containing the final states of $\mathcal{A}^2$. In this way we remove the terms, and only the terms (by Property 1), accepted by $\mathcal{A}^2$. The availability of a practical algorithm for determinisation is what makes this construction of the difference FTA feasible.

Definition 5 (Construction of difference of FTAs)**.** *Let $\mathcal{A}^1, \mathcal{A}^2$ be FTAs $(Q^1, Q^1_f, \Sigma, \Delta^1)$ and $(Q^2, Q^2_f, \Sigma, \Delta^2)$ respectively. Let $(Q', Q'_f, \Sigma, \Delta')$ be the determinisation of $\mathcal{A}^1 \cup \mathcal{A}^2$. Let $Q^2 = \{Q' \in Q' \mid Q' \cap Q^2_f \neq \emptyset\}$. Then $\mathcal{A}^1 \setminus \mathcal{A}^2 = (Q', Q'_f \setminus Q^2, \Sigma, \Delta')$.*

Next we introduce a new operation over FTA called *state splitting*, which consists of splitting a state q into a number of states, based on a partition of the set of transitions whose rhs is q. We define this splitting as follows:

Definition 6 (Splitting a state in an FTA)**.** *Let $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ be an FTA. Let $q \in Q$ and $\Delta_q = \{t \in \Delta \mid \mathsf{rhs}(t) = q\}$. Let $\Phi = \{\Delta^1_q, \ldots, \Delta^k_q\}$ $(k > 1)$ be some partition of Δ_q. Introduce k new states $q_1, \ldots, q_k$. Then the FTA $\mathsf{split}_\Phi(\mathcal{A})$ is $(Q^s, Q^s_f, \Sigma, \Delta^s)$ where:*

- $Q^s = Q \setminus \{q\} \cup \{q_1, \ldots, q_k\}$;
- $Q^s_f = Q_f \setminus \{q\} \cup \{q_1, \ldots, q_k\}$ *if* $q \in Q_f$, *otherwise* $Q^s_f = Q_f$;
- $\Delta^s = \mathsf{unfold}_q(\Delta \setminus \Delta_q \cup \{\mathsf{lhs}(t) \to q_i \mid t \in \Delta^i_q, i = 1..k\})$, *where* $\mathsf{unfold}_q(\Delta')$ *is the result of repeatedly replacing a transition $f(\ldots, q, \ldots) \to s \in \Delta'$ by the set of k transitions $\{f(\ldots, q_1, \ldots) \to s, \ldots, f(\ldots, q_k, \ldots) \to s\}$ until no more such replacements can be made.*

Proposition 1. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.

Proof. Let $\mathcal{A} = \langle Q, Q_f, \Sigma, \Delta\rangle$ and $\mathsf{split}_\Phi(\mathcal{A}) = \langle Q^s, Q^s_f, \Sigma, \Delta^s\rangle$. Let $\mathsf{split}(q)$ mean that the state q is split and let $q_1, \ldots, q_k$ be the new states introduced during splitting. We write $\Rightarrow^*$ for derivations in $\mathcal{A}$ and $\Rightarrow^*_s$ for derivations in $\mathsf{split}_\Phi(\mathcal{A})$. We first prove by induction on the depth of terms that for all terms t and states $q \in Q$,

$$(\mathsf{split}(q) \to (t \Rightarrow^* q \equiv \exists i.(t \Rightarrow^*_s q_i))) \wedge (\neg\mathsf{split}(q) \to (t \Rightarrow^* q \equiv t \Rightarrow^*_s q)). \tag{1}$$

Base case. Let a be a term of depth 1.

$$\begin{array}{lll}
\mathsf{split}(q) \to & & \\
a \Rightarrow^* q & \equiv & a \to q \in \Delta \ \wedge\ \exists\, i.(q \in \Delta^i_q \wedge a \to q_i \in \Delta^s) \\
 & & \text{following Definition 6} \\
 & \equiv & \exists i.(a \Rightarrow^*_s q_i) \\
\neg\mathsf{split}(q) \to & & \\
a \Rightarrow^* q & \equiv & a \to q \in \Delta \ \wedge\ a \to q \in \Delta^s \\
 & \equiv & a \Rightarrow^*_s q)
\end{array}$$

Inductive case. Let $f(t_1, \ldots, t_n) \Rightarrow q$ where $f(t_1, \ldots, t_n)$ is a term of depth $k + 1$ and assume that the property holds for all terms with depth at most k.

$\mathsf{split}(q) \rightarrow$
$f(t_1,\ldots,t_n) \Rightarrow^* q \equiv \exists\, r_1,\ldots,r_n\,.(f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge$
$t_1 \Rightarrow^* r_1 \wedge \cdots \wedge t_n \Rightarrow^* r_n) \wedge$
$(\mathsf{split}(r_1) \vee \neg\mathsf{split}(r_1)) \wedge \cdots \wedge (\mathsf{split}(r_n) \vee \neg\mathsf{split}(r_n))$
$\equiv \exists\, r_1,\ldots,r_n\,.[f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge$
$(\mathsf{split}(r_1) \wedge \exists i_1(t_1 \Rightarrow^*_s r_{1,i_1})) \vee (\neg\mathsf{split}(r_1) \wedge t_1 \Rightarrow^*_s r_1) \wedge$
$\ldots$
$(\mathsf{split}(r_n) \wedge \exists i_n(t_n \Rightarrow^*_s r_{n,i_n})) \vee (\neg\mathsf{split}(r_n) \wedge t_n \Rightarrow^*_s r_n))]$
by inductive hypothesis after rearranging formula, since $t_1,\ldots,t_n$ have depth at most k
$\equiv \exists\, r_1,\ldots,r_n\, \exists i_1,\ldots,i_n.[f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge$
$((t_1 \Rightarrow^*_s r_{1,i_1} \vee t_1 \Rightarrow^*_s r_1) \wedge \cdots \wedge (t_n \Rightarrow^*_s r_{n,i_n} \vee t_n \Rightarrow^*_s r_n))]$
after rearranging formula, moving quantifiers outwards
and eliminating $\mathsf{split}(r_i) \vee \neg\mathsf{split}(r_i)$, $1 \le i \le n$
$\equiv \exists\, i, r_1,\ldots,r_n\, \exists i_1,\ldots,i_n.[f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i \in \Delta^s \wedge$
$((t_1 \Rightarrow^*_s r_{1,i_1} \vee (r_{1,i_1} = r_1 \wedge t_1 \Rightarrow^*_s r_1)) \wedge \cdots \wedge (t_n \Rightarrow^*_s r_{n,i_n} \vee (r_{n,i_n} = r_n \wedge t_n \Rightarrow^*_s r_n)))]$
applying Definition 6 to introduce $f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i$
since $f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i$ is included after applying unfold to $f(r_1,\ldots,r_n) \rightarrow q_i$
$\equiv \exists\, i.f(t_1,\ldots,t_n) \Rightarrow^*_s q_i$
$\neg\mathsf{split}(q) \rightarrow$
$f(t_1,\ldots,t_n) \Rightarrow^* q \equiv$ [Similar to previous case but where $f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q$
is in the unfolding of $f(r_1,\ldots,r_n) \rightarrow q$ in Definition 6]
$\equiv f(t_1,\ldots,t_n) \Rightarrow^*_s q$

Finally, if $q \in Q_f$ and $\mathsf{split}(q)$ then $q_i \in Q^s_f$ where q_i is a new state introduced during the splitting. It follows from this and property (1) that for all t, $\exists q \in Q_f\,.\, t \Rightarrow^* q \equiv \exists q' \in Q^s_f\,.\, t \Rightarrow^*_s q'$. Thus for all t, $t \in \mathcal{L}(\mathcal{A})$ iff $t \in \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.
□

4. Horn clauses and their trace automata

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \ge 0)$, where ϕ is a conjunction of constraints with respect to some constraint theory, X_i, 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 $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body.

There is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. They are also sometimes referred to as negative clauses. We follow the syntactic conventions of constraint logic programs and write a clause as $p(X) \leftarrow \phi, p_1(X_1),\ldots,p_k(X_k)$.

4.1. Interpretations and models

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z_1,\ldots,Z_n)$ where $Z_1,\ldots,Z_n$ are distinct variables and ϕ is a constraint over $Z_1,\ldots,Z_n$. This set may be infinite. The constrained fact $A \leftarrow \phi$ is shorthand for the set of variable-free facts $A\theta$ such that $\phi\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies (whenever the body of a clause holds under the given interpretation then so does the head) each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is a satisfiable set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator (called S^D_P in [12, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint theory D. Furthermore $\mathsf{lfp}(S^D_P)$ can be computed

as the limit of the ascending sequence of interpretations $\emptyset, S^D_P(\emptyset), S^D_P(S^D_P(\emptyset)), \ldots$. This sequence provides a basis for abstract interpretation of CHC clauses. The minimal model of P is equivalent to the set of atomic logic consequences of P.

4.2. The constrained Horn clause verification problem.

Given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the derivability of the predicate false. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F.

Lemma 1. *P has a model if and only if $P \not\models \mathsf{false}$.*

Proof. Writing $I(F)$ to mean that interpretation I satisfies F, we have:

$$\begin{array}{rcl} P \not\models \mathsf{false} & \equiv & \text{there exists an interpretation } I \text{ such that } I(P) \text{ and } \neg I(\mathsf{false}) \\ & & \textit{by definition of } \models \\ & \equiv & \text{there exists an interpretation } I \text{ such that } I(P) \\ & & (\textit{since } \neg I(\mathsf{false}) \textit{ is true by defn. of } \mathsf{false}) \\ & \equiv & P \text{ has a model.} \end{array}$$

□

This lemma holds for arbitrary interpretations (only assuming that the predicate false is interpreted as false), uses only the textbook definitions of "interpretation" and "model" and does not depend on the constraint theory. We have yet another equivalent formulation of the CHC verification problem.

Lemma 2. *P has a model if and only if $\mathsf{false} \notin M[\![P]\!]$.*

Proof. Follows from the equivalence of the minimal model of P with the set of atomic logical consequences of P [13]. See also Proposition 2 later. □

It is this formulation that is most relevant to our method, since we compute over-approximations of $M[\![P]\!]$ by abstract interpretation. That is, if $\mathsf{false} \notin M'$ where $M[\![P]\!] \subseteq M'$ then we have shown that P has a model. An assertion ϕ is an invariant (over-approximation) for a predicate q in P, if $P \models \forall(q \rightarrow \phi)$. If a set of Horn clauses P have a model then we say that P is *safe*, otherwise we say that P is *unsafe*.

4.3. Trace automata for CHCs

Before constructing the trace automaton we introduce identifiers for each clause. An identifier is a function symbol whose arity is the same as the number of atoms in the clause body. For instance a clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ is assigned a function symbol with arity k. More than one clause can be assigned the same function symbol, but all the clauses with the same identifier have the same structure, including their constraints; that is, they differ only in one or more predicate names. Given a set of CHCs and a set Σ of ranked function symbols, let $\mathsf{id}_P : P \rightarrow \Sigma$ be the assignment of function symbols to clauses.

Definition 7 (Trace FTA for a set of CHCs)**.** *Let P be a set of CHCs. Define the trace FTA for P as $\mathcal{A}_P = (Q, Q_f, \Sigma, \Delta)$ where*

- *Q is the set of predicate symbols of P;*
- *$Q_f = Q$;*
- *Σ is a set of function symbols;*
- *$\Delta = \{c(p_1, \ldots, p_k) \rightarrow p \mid \text{where } c \in \Sigma,\ c = \mathsf{id}_P(cl), \text{where } cl = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)\}$.*

The elements of $\mathcal{L}(\mathcal{A}_P)$ are called trace terms for P. In Section 5 we will see that several clauses differing only in their predicate names are assigned the same function symbol.

Example 1. *Let P be the set of CHCs in Figure 1. Let id_P map the clauses to $c_1, \ldots, c_4$ respectively. Then $\mathcal{A}_P = (Q, Q_f, \Sigma, \Delta)$ where:*

$$\begin{array}{lcllcl} Q & = & \{\texttt{mc91}, \texttt{false}\} & \Delta & = & \{c_1 \to \texttt{mc91}, \\ Q_f & = & \{\texttt{mc91}, \texttt{false}\} & & & c_2(\texttt{mc91}, \texttt{mc91}) \to \texttt{mc91}, \\ \Sigma & = & \{c_1, c_2, c_3, c_4\} & & & c_3(\texttt{mc91}) \to \texttt{false},\ c_4(\texttt{mc91}) \to \texttt{false}\} \end{array}$$

For each trace term there exists a corresponding derivation tree called an AND-tree, which is unique up to variable renaming. The concept of an AND-tree is derived from [14] and [15].

Definition 8 (AND-tree for a trace term)**.** *Let P be a set of CHCs and let $t \in \mathcal{L}(\mathcal{A}_P)$. Denote by $\mathsf{AND}(t)$ the following labelled tree, where each node of $\mathsf{AND}(t)$ is labelled by a clause and an atomic formula.*

1. *For each subterm $c_j(t_1, \ldots, t_k)$ of t there is a corresponding node in $\mathsf{AND}(t)$ labelled by an atom $p(X)$ and a clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ which is a renamed version of some clause c in P, such that $c_j = \mathsf{id}_P(c)$; the node's children (if $k > 0$) are the nodes corresponding to $t_1, \ldots, t_k$ and are labelled by $p_1(X_1), \ldots, p_k(X_k)$.*
2. *The variables in the labels are chosen such that if a node n is labelled by a clause, the local variables in the clause body do not occur outside the subtree rooted at n.*

Definition 9 (Trace constraints)**.** *Let P be a set of CHCs. The set of constraints of a trace $t \in \mathcal{L}(\mathcal{A}_P)$, represented as $\mathsf{constr}(t)$ is the set of all constraints in the clause labels of $\mathsf{AND}(t)$.*

Definition 10 (Feasible trace)**.** *We say that a trace term t is feasible if $\mathsf{constr}(t)$ is satisfiable.*

Definition 11 (FTA for a trace term)**.** *Let P be a set of CHCs and $t \in \mathcal{L}(\mathcal{A}_P)$. The FTA A_t (whose construction is trivial) such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$ is called* the FTA for *t. The states of $\mathcal{A}_t$ are chosen to be disjoint from those of $\mathcal{A}_P$.*

Example 2 (Trace FTA)**.** *Consider the FTA in Example 1. Let $t = c_3(c_2(c_1, c_1))$. Each $\mathtt{e_i}$ represents a label in the trace. Then $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ is defined as:*

$$\begin{array}{lcl} Q & = & \{\mathtt{e}_1, \mathtt{e}_2, \mathtt{e}_3, \mathtt{e}_4\} \\ Q_f & = & \{\mathtt{e}_1\} \\ \Sigma & = & \{c_1, c_2, c_3, c_4\} \\ \Delta & = & \{c_1 \to \mathtt{e}_3,\ c_1 \to \mathtt{e}_4,\ c_2(\mathtt{e}_3, \mathtt{e}_4) \to \mathtt{e}_2, \\ & & \ c_3(\mathtt{e}_2) \to \mathtt{e}_1\} \end{array}$$

and Σ is the same as in $\mathcal{A}_P$. The trace t is not feasible since

$$\mathsf{constr}(t) = \{\mathtt{A} \leq \mathtt{100}, \mathtt{B} > \mathtt{91}, \mathtt{A} \leq \mathtt{100}, \mathtt{C} = \mathtt{A} + \mathtt{11}, \mathtt{C} > \mathtt{100}, \mathtt{D} = \mathtt{C} - \mathtt{10}, \mathtt{D} > \mathtt{100}, \mathtt{B} = \mathtt{D} - \mathtt{10}\}$$

and this is not satisfiable.

Example 3 (Trace FTA of a linear trace)**.** *Consider the FTA in Example 1 and a linear trace $t = c_3(c_1)$. Let $\mathtt{e}_1$ and $\mathtt{e}$ represents the labels in the trace. Then $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ is defined as:*

$$\begin{array}{lcl} Q & = & \{\mathtt{e}, \mathtt{e}_1\} \\ Q_f & = & \{\mathtt{e}\} \\ \Sigma & = & \{c_1, c_2, c_3, c_4\} \\ \Delta & = & \{c_1 \to \mathtt{e},\ c_3(\mathtt{e}_1) \to \mathtt{e}\} \end{array}$$

and Σ is the same as in $\mathcal{A}_P$. The trace t is not feasible.

Definition 12 (Constrained trace atom)**.** *Let P be a set of CHCs and $t \in \mathcal{L}(\mathcal{A}_P)$. Let $p(X)$ be the atom labeling the root of $\mathsf{AND}(t)$. Then the constrained trace atom of t is $\forall X.(\exists \bar{Z}.\mathsf{constr}(t) \to p(X))$, where $\bar{Z} = \mathsf{vars}(\mathsf{constr}(t)) \setminus X$.*

We now restate standard soundness and completeness results from constraint logic programming [13] in terms of the concepts defined above. We assume that the underlying constraint theory $\mathcal{T}$ has a complete satisfiability procedure. Note that the domain of linear arithmetic constraints, which is used in our experiments, satisfies these conditions.

Proposition 2. *Let P be a set of CHCs, whose underlying constraint theory $\mathcal{T}$ has a complete satisfiability procedure. Let $\mathcal{A}_P$ be the trace FTA for P. Then*

1. *for all $t \in \mathcal{L}(\mathcal{A}_P)$, $P \cup \mathcal{T} \models A$, where A is the constrained trace atom of t;*
2. *$p(c)$ is a variable-free atomic formula such that $P \cup \mathcal{T} \models p(c)$, iff*
   (a) *there exists a feasible trace $t \in \mathcal{L}(\mathcal{A}_P)$ whose constrained trace atom is of the form $\forall X.\phi \rightarrow p(X)$ where the constraint $\phi[X/c]$ is true; and*
   (b) *$p(c)$ is in $M[\![P]\!]$, the minimal model of P.*

Proof. The proof depends on a close correspondence between AND-trees (Definition 8) and derivations defined as sequences in [13]; we do not elaborate the correspondence in detail but just note that for each AND-tree with constrained trace atom $\forall X.(\exists \bar{Z}.\mathsf{constr}(t) \rightarrow p(X))$ there exists one or more derivations for $p(X)$ with answer constraint $\exists \bar{Z}.\mathsf{constr}(t)$. Conversely for each derivation for $p(X)$ with answer constraint ϕ there exists a unique AND-tree whose root is labelled with $p(X)$ and whose constrained trace atom is $\forall X.\phi \rightarrow p(X)$.

(1). Let $\mathsf{AND}(t)$ be the AND-tree for t (Definition 8), let $p(X)$ be the atom labeling the root and let $\forall X.\phi \rightarrow p(X)$ be the constrained trace atom for t.
   $\Rightarrow$ there is some derivation (as defined in [13]) for $p(X)$ having answer constraint $\phi \rightarrow p(X)$;
   $\Rightarrow$ $P \cup \mathcal{T} \models \phi \rightarrow p(X)$ by [13] (Theorem 6.0.1, Part 2).

2(a). $P \cup \mathcal{T} \models p(c)$ is equivalent to $P \cup \mathcal{T} \models X = c \rightarrow p(X)$.
   $\Rightarrow$ there is a derivation for $p(X)$ with answer constraint ϕ,
   where $\mathcal{T} \models X = c \rightarrow \phi$ (by [13] (Theorem 6.0.1, Part 4));
   $\Rightarrow$ there is a trace term t and AND-tree $\mathsf{AND}(t)$
   with root labelled by $p(X)$ and constrained trace atom $\forall X.\phi \rightarrow p(X)$, where $\mathcal{T} \models \phi[X/c]$.

2(b). Follows directly from [13] (Theorem 6.0.1, Part 1,2)

□

For part 2(a) of the proof, note that the constrained trace atom in the AND-tree can be more general than the atom $p(c)$. For example, say that $p(1)$ is a consequence of the set of CHCs; then the constrained trace atom could be $\forall X.X \geq 0 \rightarrow p(X)$.

Proposition 2 establishes the correspondence between the semantics of CHCs and the feasible traces of the trace FTA for the CHCs. Essentially, the set of feasible traces of the FTA is a representation of the minimal model of the clauses.

If we transform $\mathcal{A}_P$ to another FTA while preserving the set of traces, we also preserve the feasible traces. More generally, we can transform $\mathcal{A}_P$ to another FTA $\mathcal{A}'$ so long as $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A}_P)$ and the elements of $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{L}(\mathcal{A}')$ are all infeasible. In this case the feasible traces of $\mathcal{L}(\mathcal{A}')$ are still a representation of the minimal model of P. We will exploit this in our refinement procedure (see Section 5).

4.4. Generation of CHCs from a trace FTA

Now we describe a procedure (Algorithm 2) for generating a set of clauses P' from an FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ and a set of clauses P. We assume that Σ is the same as that of $\mathcal{A}_P$; so Σ is the range of the function id_P mapping clauses of P to function symbols. The transitions Δ are not in product form; a modification of the algorithm and its correctness proposition is possible for product form (which is in fact an enabling factor which makes possible the determinisation of FTAs in practice) but we omit that here for the simplicity of presentation. We first introduce an injective function for renaming the states of $\mathcal{A}$ since we need predicate names for the generated clauses.

$$\rho : Q \rightarrow \mathsf{Predicates}$$

The function ρ maps each FTA state to a distinct predicate name. The algorithm simply generates a clause for each transition, applying the renaming function from states to predicates, and introducing variables arguments according to the pattern obtained from any clause with the corresponding identifier (all clauses with the same identifier having the same variable pattern).

Apart from generating a set of clauses P', Algorithm 2 also generates the clause identification mapping $\mathsf{id}_{P'}$, preserving the function symbols from the FTA. In this way the set of traces is preserved from P to P'. The correctness of Algorithm 2 is expressed by the following proposition.

Input: An FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$, set of Horn clauses P, injective functions $\rho : Q \to \mathsf{Predicates}$, $\mathsf{id}_P : P \to \Sigma$
Output: A set of Horn clauses P' represented by $\mathcal{A}$ and function $\mathsf{id}_{P'} : P' \to \Sigma$
$P' \leftarrow \emptyset$;
for *each* $c_i(q_1, \ldots, q_n) \to q$ *(where* $n \geq 0$*)* $\in \Delta$ **do**
  let $c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$ be the clause in P where $\mathsf{id}_P(c) = c_i$;
  $c_{new} = \rho(q)(X) \leftarrow \phi, \rho(q_1)(X_1), \ldots, \rho(q_n)(X_n)$;
  $\mathsf{id}_{P'}(c_{new}) = c_i$;
  $P' \leftarrow P' \cup \{c_{new}\}$;
end
return P';

Algorithm 2: Generating a set of clauses represented by an FTA

Proposition 3. *Let P be a set of CHCs and let $\mathcal{A}$ be an FTA whose signature is the same as that of $\mathcal{A}_P$. Let P' be the set of clauses generated from $\mathcal{A}$ and P by Algorithm 2. Then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P'})$. Furthermore if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_P)$ and $\mathcal{L}(\mathcal{A})$ includes all the feasible traces of $\mathcal{L}(\mathcal{A}_P)$ then the minimal model of P' is the same as the minimal model of P, modulo predicate renaming.*

Proof. We first prove that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P'})$, that is, $\forall t \,.\, t \in \mathcal{L}(\mathcal{A}) \equiv t \in \mathcal{L}(\mathcal{A}_{P'})$. The proof is by induction on the depth of t. Let $\mathcal{A} = \langle Q, Q_f, \Sigma, \Delta \rangle$ and $\mathcal{A}_{P'} = \langle Q', Q'_f, \Sigma, \Delta' \rangle$ and we assume that $Q = Q_f$ and $Q' = Q'_f$.

- Base case.

$$\begin{array}{rcl} t \text{ has depth 0 and } t \in \mathcal{L}(\mathcal{A}) & \equiv & \exists\, t \to q \in \Delta \\ & \equiv & \exists\, c = p(X) \leftarrow \phi \in P \textit{ where } \mathsf{id}_P(c) = t \\ & \equiv & \exists\, c_{new} = \rho(q)(X) \leftarrow \phi \in P' \textit{ and } \mathsf{id}_{P'}(c_{new}) = t \\ & \equiv & \exists\, t \to \rho(q) \in \Delta' \\ & \equiv & t \in \mathcal{L}(\mathcal{A}_{P'}) \end{array}$$

- Inductive case. Assume that for all terms t of depth at most k, $t \Rightarrow^* q$ in $\mathcal{A}$ iff $t \Rightarrow^* \rho(q)$ in $\mathcal{A}_{P'}$. Let $t = c_t(t_1, \ldots, t_n)$ have depth $k + 1$.

$$\begin{array}{rcl} c_t(t_1, \ldots, t_n) \in \mathcal{L}(\mathcal{A}) & \equiv & \exists\, c_t(q_1, \ldots, q_n) \to q \in \Delta \;\wedge\; t_i \Rightarrow^* q_i, 1 \leq i \leq n \\ & \equiv & \exists\, c_t(q_1, \ldots, q_n) \to q \in \Delta \;\wedge\; t_i \Rightarrow^* \rho(q_i), 1 \leq i \leq n \\ & & \text{by ind. hyp. since depth of } t_1, \ldots t_n \text{ is at most } k \\ & \equiv & \exists\, c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n) \in P \textit{ where } \mathsf{id}_P(c) = c_t \\ & & \wedge\; t_i \Rightarrow^* \rho(q_i), 1 \leq i \leq n \\ & \equiv & \exists\, c_{new} = \rho(q)(X) \leftarrow \phi, \rho(q_1)(X_1), \ldots, \rho(q_n)(X_n) \in P' \textit{ and } \mathsf{id}_{P'}(c_{new}) = c_t \\ & & \wedge\; t_i \Rightarrow^* \rho(q_i), 1 \leq i \leq n \\ & \equiv & \exists\, c_t(\rho(q_1), \ldots, \rho(q_n)) \to \rho(q) \in \Delta' \\ & & \wedge\; t_i \Rightarrow^* \rho(q_i), 1 \leq i \leq n \\ & \equiv & c_t(t_1, \ldots, t_n) \in \mathcal{L}(\mathcal{A}_{P'}) \end{array}$$

This completes the proof that $\mathcal{L}(\mathcal{A}_{P'}) = \mathcal{L}(\mathcal{A})$. Now assume that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_P)$ and includes all the feasible traces of $\mathcal{L}(\mathcal{A}_P)$; that is, for all feasible traces t, $t \in \mathcal{L}(\mathcal{A}_P)$ iff $t \in \mathcal{L}(\mathcal{A})$. Then by Proposition 2 $M[\![P]\!] = M[\![P']\!]$ since $\mathcal{A}$ is the trace FTA for P'. □

Example 4 (Generation of clauses from an FTA)**.** *Consider the following transitions, relating to the signature for the program in Figure 1. The set of states is* {`[false]`, `[mc91]`, `[e,false]`, `[mc91,e1]`}*. (These are elements of the powerset of the set of states* {`false`,`mc91`,`e`,`e1`}*obtained from the union of FTA in Example 1 and FTA in Example 3, which were generated by the determinisation algorithm).*

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
```

```
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
c3([mc91]) -> [false].
c4([mc91, e1]) -> [false].
c4([mc91]) -> [false].
c3([mc91, e1]) -> [e, false].
```

The clauses generated by Algorithm 2 are the following, with the renaming function $\rho = \{[\texttt{false}] \mapsto \texttt{false}, [\texttt{mc91}] \mapsto \texttt{mc91}, [\texttt{e}, \texttt{false}] \mapsto \texttt{false\_1}, [\texttt{mc91}, \texttt{e1}] \mapsto \texttt{mc91\_1}\}$. *Below we also show the clause identifiers (the* id *function for the generated clauses) showing that several clauses can have the same identifier, thus preserving traces.*

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
c3: false_1 :- A =< 100, B > 91, mc91_1(A,B).
```

4.5. Abstract Interpretation of Constrained Horn Clauses

Abstract interpretation [16] is a technique which derives sound over-approximations by computing abstract fixed points. Convex polyhedron analysis (CPA) [17] is a program analysis technique based on abstract interpretation [16]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CHCs was by Benoy and King [18].

We summarise briefly the elements of convex polyhedron analysis for CHC; further details (with application to CHC) can be found in [17, 18]. The abstract interpretation consists of the computation of an increasing sequence of elements of the abstract domain of tuples of convex polyhedra (one for each predicate) $\mathcal{D}^n$. We construct a monotonic *abstract semantic function* $F_P : \mathcal{D}^n \rightarrow \mathcal{D}^n$ for the set of Horn clauses P, approximating the concrete semantic "immediate consequences" operator.

Since $\mathcal{D}^n$ contains infinite increasing chains, a *widening* operator for convex polyhedra [17] is needed to ensure convergence of the sequence. The sequence computed is $Z_0 = \bot^n$, $Z_{n+1} = Z_n \nabla F_P(Z_n)$ where ∇ is a widening operator for convex polyhedra and the empty polyhedron is denoted $\bot$. The conditions on ∇ ensure that the sequence stabilises; thus for some finite j, $Z_i = Z_j$ for all $i > j$ and furthermore the value Z_j represents an over-approximation of the least model of P. Algorithm 3 presents convex polyhedral analysis for Horn clauses. For each constrained fact derived from the set of clauses P using this algorithm, there is a derivation tree (trace term) associated with it. These are syntactically possible trace terms using the clauses in P but may not be feasible due to abstraction. Thus all such trace terms are in $\mathcal{L}(\mathcal{A}_P)$.

Much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [19]. A threshold is an assertion that is combined with a widening operator to improve its precision.

Our tool for convex polyhedral abstract interpretation, called CPA in the rest of this paper, uses the Parma Polyhedra Library [20] to implement the operations on convex polyhedra, and incorporates a threshold generation phase based on the method described by Lakhdar-Chaouch *et al.* [21], as well as a constraint strengthening pre-processing which propagates constraints both forwards and backwards in the clauses of P.

4.5.1. Computing thresholds for widening

Recently, a technique for deriving more effective thresholds was developed [21], which we have adapted and found to be effective in experimental studies. In brief, the method collects constraints by iterating the concrete imme-

Input: CHCs P
Output: a set of constrained facts Z_i
$i \leftarrow 0$;
$Z_0 \leftarrow \bot$;
$New \leftarrow \bot$;
repeat
  forall the $p(X) \leftarrow Body \in P$ **do**
    $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
  end
  $Z_{i+1} \leftarrow Z_i \nabla (New \sqcup Z_i)$ ▷ Upper bound and widen ;
  $i \leftarrow i + 1$;
until $Z_i \sqsubseteq Z_{i-1}$;
return Z_i ;

Algorithm 3: Algorithm for convex polyhedral abstraction

diate consequence function S_P^D three times starting from the "top" interpretation, that is, the interpretation in which all atomic facts are true. The thresholds are computed by the following method. Let S_P^D be the standard immediate consequence operator for CHCs mentioned in Section 4.1. That is, if I is a set of constrained facts, $S_P^D(I)$ is the set of constrained facts that can be derived in one step from I. Given a constrained fact $p(\bar{Z}) \leftarrow C$, define $\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow C)$ to be the set of constrained facts $\{p(\bar{Z}) \leftarrow C_i \mid C = C_1 \wedge \ldots \wedge C_k, (1 \leq i \leq k)\}$. The function $\mathsf{atomconstraints}$ is extended to interpretations by $\mathsf{atomconstraints}(I) = \bigcup_{p(\bar{Z}) \leftarrow C \in I} \{\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow C)\}$.

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(\bar{Z}) \leftarrow \mathsf{true}$ for each predicate p. We perform three iterations of S_P^D (represented as $S_P^{D(3)}$) starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, $\mathsf{thresholds}$ is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(S_P^{D(3)}(I_\top))$$

A difference from the method in [21] is that we use the concrete semantic function S_P^D rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See [21] for further details. Threshold constraints that are not invariants are simply discarded during widening. Thresholds constraints are not necessarily over-approximations (invariants).

The operation $\mathsf{thresholds}(P)$ can become expensive and generate a very large number of constraints. Alternatively we can generate more general threshold constraints (called abstract threshold), possibly losing precision while gaining efficiency, by following more closely the approach defined in [21], using the abstract semantic function F_P. Then the threshold operation P becomes

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(F_P^{(3)}(I_\top))$$

4.5.2. Pre-processing of Horn clauses by specialisation

The effectiveness of abstract interpretation can be improved by combining it with specialisation with respect to some property. Therefore we specialise (pre-process) the set of clauses with respect to the property to be verified using the method described in [22]. The method is summarised as follows: the inputs are a set of CHCs P and an atomic formula A (a property) and the output is P_A, a set of specialised clauses.

1. Compute a *query-answer transformation* [22] of P with respect to A, denoted P^{qa}, containing predicates p^{q} and p^{a} representing query and answer predicates for each predicate p in P.
2. Compute an over-approximation M of the model of P^{qa} using abstract interpretation.
3. Strengthen the constraints in the clauses in P, by adding constraints from the answer predicates in M. That is, for each clause

$$p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$$

in P, we construct a clause

$$p(X) \leftarrow \phi, \phi_0, \phi_1, \ldots, \phi_n, p_1(X_1), \ldots, p_k(X_k)$$

in P_A, where $p^{\mathsf{a}}(X) \leftarrow \phi_0$, $p_1^{\mathsf{a}}(X) \leftarrow \phi_1, \ldots, p_n^{\mathsf{a}}(X) \leftarrow \phi_n$ are in M'.

The method propagates constraints globally, both forwards and backwards, and makes explicit constraints from the original clauses. This allows better analysis of the transformed clauses. Furthermore, the method is independent of the abstract domain and the constraints theory underlying the clauses.

5. Refinement of Horn clauses using trace automata

If an over-approximation of the clauses derived by polyhedral abstraction does not contain false, the clauses are safe. However if false is contained in the approximation, we do not know whether the clauses are unsafe or whether the approximation was too imprecise. In such cases we can produce a trace term $t \in \mathcal{A}_P$ using the clauses in P which justifies the abstract derivation of false. The feasibility of this trace can be checked by a constraint satisfiability check. If the trace is feasible, then it corresponds to a proof of unsafety. Otherwise, refinement is considered based on this trace. In some other approaches, a more precise abstract domain is derived from the trace. In our refinement approach, which is described next, we aim to generate a modified set of clauses that could yield a better approximation. This is achieved through the steps shown in Algorithm 4.

Input: A set of Horn clauses P and an infeasible trace $t \in \mathcal{A}_P$
Output: A set of refined Horn clauses P'

1. construct the trace FTA $\mathcal{A}_P$ (Definition 7);
2. construct an FTA $\mathcal{A}_t$ such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$ (Definition 11);
3. compute the difference FTA $\mathcal{A}_P \setminus \mathcal{A}_t$ (Definition 5);
4. generate P' from $\mathcal{A}_P \setminus \mathcal{A}_t$ and P (Algorithm 2) ;

return P';

Algorithm 4: Algorithm for clause refinement using FTA operations

Both $\mathcal{A}_P$ and $\mathcal{A}_t$ in Algorithm 4 are deterministic FTAs by construction, however their union is not. Determinisation is used to generate the difference FTA (step 3) and its result is in product form. The program P' has the same model (modulo predicate renaming) as P, since the steps result in the removal of an infeasible trace but all other traces are preserved.

Removal of one trace from the clauses might not seem much of a refinement. However, modifying the clauses to remove a single trace can result in significant restructuring, which arises as a side-effect of determinisation which isolates the infeasible trace. This in turn can induce a more precise abstract interpretation, with less precision loss due to convex hull operations and widening.

The correctness of this refinement follows from Proposition 3. In particular false $\in M[\![P]\!]$ if and only if false $\in M[\![P']\!]$ (assuming that the predicate renaming at least preserves the predicate name false).

Example 5. *Consider again the FTA shown in Example 4. This is in fact the determinisation of $\mathcal{A}_P \cup \mathcal{A}_t$ where P is the set of clauses in Figure 1 and $\mathcal{A}_t$ where t is the infeasible trace* `c3(c1)`*. The only accepting state of $\mathcal{A}_t$ is* `e`*; thus to construct the difference $\mathcal{A}_P \setminus \mathcal{A}_t$ we need only to remove from the automaton the states containing* `e`*, namely* `[mc91,e]`*. We can also remove any transitions containing this state in the right hand side. This leaves the following FTA and refined program in Figure 2, using the same renaming function as in Example 4. In this program, the infeasible trace corresponding to* `c3(c1)` *cannot be constructed.*

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
```

```
c3([mc91]) -> [false].
c4([mc91]) -> [false].
c4([mc91, e1]) -> [false].
```

It can be seen that although the infeasible trace was very simple, its removal led to a considerably restructured set of clauses. We have not shown the product form here, which is in fact somewhat more compact.

The refinement process guarantees progress; that is, the infeasible computation once eliminated never arises again. Due to the construction of the id mapping for P' the traces in the languages of the FTAs of P and P' are preserved, apart from the eliminated trace.

Proposition 4 (Progress)**.** *Let P be a set of CHCs, and t be a trace in P. Let P' be a refined set of CHCs obtained from P after the removal of t. Then t cannot be generated in any approximation of P'.*

Proof. The proof of this proposition follows from the following points:

1. $\mathcal{A}_P$ recognizes all syntactically possible traces of P, which is an over-approximation of the traces of P since the constraints in P are not taken in account while constructing $\mathcal{A}_P$.

2. After the removal of the trace t from all possible traces of P (step 3 of Algorithm 4) the language of $\mathcal{A}_P \setminus \mathcal{A}_t$ does not contain t (difference automata).

3. Then using Algorithm 2 to generate P' from $\mathcal{A}_P \setminus \mathcal{A}_t$ and P, t will be syntactically impossible trace in P' (follows from Proposition 3).

4. Since t is syntactically impossible trace in P', there is no constrained fact associated with it in any abstract domain using P' (see Section 4.5). □

Progress is an interesting and relevant refinement property but it gives no guarantees that a proof will eventually be found if such exists. In the worst case the algorithm will just eliminate longer and longer infeasible traces. Even if there exists some convex polyhedral approximation that establishes P's satisfiability, the abstract interpretation algorithm involving the widening heuristic cannot guarantee to find it.

5.1. Further refinement: splitting a state in the trace FTA

We also apply a tree-automata-based transformation to split states representing predicates where convex hull operations have lost precision. A typical case is where a number of clauses with the same head predicate contain disjoint constraints, such as a predicate representing an if-then-else statement in an imperative program. The clauses defining the statement will have a clause for the *then* branch and a clause for the *else* branch. The respective constraints in these clauses are disjoint since one is the negation of the other. The convex hull will thus contain the whole space for the variables involved in these constraints.

As defined in Definition 6, the FTA state corresponding to such a predicate can be split. We partition the transitions corresponding to the clauses according to the disjoint groups of constraints and apply the procedure in Definition 6, preserving the set of traces. Thus the feasible traces and the model of the resulting clauses is preserved. This enhances precision of polyhedral analysis [23].

Splitting has to be carried out in a controlled manner to prevent blow up in the size of FTA and hence on the size of the clauses generated. With this in mind we split only those states appearing in a counterexample trace, but this is not necessary in our approach to avoid a counterexample.

It would have been possible to formulate the splitting operation directly on CHCs, without any reference to FTAs. However, since the whole procedure is based on transformations that preserve the set of feasible traces, we preferred to present splitting as a language-preserving operation on FTAs.

6. Experiments on CHC benchmark problems

6.1. Experimental settings

Our tool consists of an implementation of a *convex polyhedra analyser* for CLP written in Ciao Prolog[1] interfaced to the Parma Polyhedra Library [20] as well as an implementation of an FTA determiniser written in Java. It takes as

[1] http://ciao-lang.org/

	CPA	CPA+R	CPA+R+Split	QARMC	VeriMAP (GenPH)	TRACER-SPost	TRACER-WPre	ELDARICA
solved (safe/unsafe)	160 (142/18)	182 (160/22)	195 (164/31)	178 (141/37)	185 (154/31)	91 (74/17)	103 (85/18)	159(135/24)
timeout or errors	56	34	22	38	38	125	113	57
average time (secs.)	5.98	51.66	50.08	59.1	57.93	305.03	225.82	50.02

Figure 4. Experimental results on 216 (179 safe / 37 unsafe) CHC verification problems with a timeout of five minutes

input a CLP program and returns "safe", "unsafe" or "unknown" (after resources are exhausted). The input is first pre-processed using the method described in 4.5.2. The benchmark set contains 216 CHCs verification problems (179 safe and 37 unsafe problems), taken mainly from the repositories of several state-of-the-art software verification tools such as DAGGER [24] (21 problems), TRACER [25] (66 problems), InvGen [26] (68 problems), and also from the TACAS 2013 Software Verification Competition [27] (52 problems). These problems are also available in `C` (`http://akira.ruc.dk/~kafle/comlan-vmcai15-benchmarks.zip`) and they were first translated to CLP form[2]. The chosen problems are representatives of different categories of the Software Verification Competition (loops, control flow and integer, SystemC etc.) as well as specific problems used to demonstrate the strength of different verification tools. The benchmarks in CLP form are available from `http://akira.ruc.dk/~kafle/VMCAI15-Benchmarks.zip`. The experiments were carried out on an Intel(R) computer with a 2.66GHz processor running Debian 5 in 6 GB memory.

6.2. Summary of results

The results of our experiments are summarised in Table 4. Column CPA summarises the results using our own *convex polyhedra analyser* (Section 4.5) with no refinement step. Column CPA+R shows the results obtained by iterating the CPA algorithm with the refinement step described in Section 5, Algorithm 4. Column CPA+R+Split incorporates the FTA-based state splitting into the refinement step (Section 5.1). In all the above cases, we used a concrete threshold generation as described in 4.5.1. Column QARMC shows the results obtained on the same problems using the QARMC tool [28, 29]. The columns VeriMAP(GenPH), TRACER-SPost, TRACER-WPre respectively report results using the VeriMAP system implementing Iterated Specialization method with the generalization operator GenPH [30], TRACER [25] using the strongest postcondition (SPost) and the weakest precondition (WPre) options. The results in these three columns are taken from [30] since we couldn't run these tools. We used the same set of benchmarks as in [30]. The last column ELDARICA reports results using Eldarica tool[3] which uses disjunctive interpolants for the Horn clause verification purpose [31].

6.3. Discussion of results

The results show that CPA is reasonably effective on its own, solving 74% (160/216) of the problems. When combined with a refinement phase we can solve further 22 problems. Although only one infeasible trace is eliminated in each refinement step, the refined program splits some of the predicates appearing in the trace, which we noted to be a crucial point of precision for polyhedral analysis [23]. When adding the state splitting refinement we solve an additional 13 problems. Further splitting would solve more problems but we are unwilling to introduce uncontrolled splitting due to the blow up in program size that could result. The maximum number of iterations required to solve a problem was 8. Although the timeout limit was five minutes, only 5% of the solved problems required more than one minute.

Our implementation uses the product form for DFTAs produced by the determinisation algorithm, although the formalisation of refinement in Section 5 uses only standard FTA transitions. Although the traces for clauses with predicates produced from product states differ from the original clauses, they can be regarded as representing the original traces, by unfolding the clauses resulting from ϵ-transitions. Product form adds to the scalability of the approach, especially for Horn clauses with more than one body atom. Empirically we have not shown this here but this is due to the scalability of the product form during determinisation of FTAs (see [11]).

[2]Thanks to Emanuele De Angelis for the translation

[3]http://lara.epfl.ch/w/eldarica

	CPA+R+Split	CPA+R+Split+AT	QARMC
solved (safe/unsafe)	114 (54/60)	114 (54/60)	110 (42/68)
timeout or errors	18	18	22
average time (secs.)	72.92	66.28	52.4

Table 1. Experimental results on 132 CHC verification problems with a timeout of five minutes

6.4. Comparison with other tools

On the set of verification problems considered, our results (*CPA+R+Split*) improve on other tools both in average time and the number of instances solved. The results of VeriMAP and QARMC are close to ours while results of TRACER is bit far. This is due to the fact that TRACER uses symbolic execution and does not scale well. Out of 216 problems QARMC solves 178 problems with an average time of 59 seconds whereas we can solve 195 problems with an average time of 50 seconds. However, all unsafe programs in the benchmark set are solved by QARMC in contrast to ours. Surprisingly enough, the number of unsafe problems solved by VeriMAP and our tool is the same. Since both of our tool use convex hull and widening, the precision lost due to these operators in the rest of the unsafe programs cannot be recovered. The model checking algorithm implemented in Eldarica for Horn solving is similar in spirit as the one described in [29] but uses disjunctive interpolation for counterexamples generalization instead of tree interpolation which is strictly more general than tree interpolation [31]. We suspect that it is due to this, the average time taken by Eldarica is slightly less than that of QARMC though it solves lesser number of instance than QARMC. Our results show that for these set of examples, tools using polyhedral abstraction seems more powerful than the others.

Convex polyhedral analysis is good at finding the required invariants to prove the safety of a program and due to this our tool and VeriMAP solved more safe problems than QARMC. On the other hand, QARMC seems to be more effective at finding bugs. Most of the problems challenging to us come from some particular categories e.g. SystemC (modeled over fixed size integers) and Control Flow and Integer Variables of [27] which requires some specific techniques to solve. Safe problems challenging to us are also challenging to QARMC though this is not the case for unsafe problems.

6.5. Additional experiments on SV-COMP-15

We chose a subset of 132 problems from SV-COMP 2015[4] [32]. This set contains benchmarks from the categories which were not reported in our experiments before such as *recursive benchmarks* which needs recursive analysis. Additionally it contains some benchmarks from *Loop* category such as *loop-acceleration, loop-lit, loop-new*. We used SeaHorn [33, 5], a verification framework based on LLVM, for Horn clause generation. SeaHorn first compiles C to LLVM intermediate representation (LLVM IR), also known as bitcode using *clang*, a C-family front-end for LLVM[5]. The bitcode is further simplified and optimized reusing the vast amount of work done on LLVM (e.g. function inlining, dead code elimination, CFG simplifications etc.) whose purpose is to make the verification task easier. The resulting bitcode is translated to Horn clauses using different semantics for example small step, large block encoding etc. More details can be found in [33, 5]. These benchmarks are available in C as well as in Horn clause form from `http://akira.ruc.dk/~kafle/comlan-vmcai15-benchmarks.zip`. The results are summarised in the Table 6.5. The column *CPA+R+Split+AT* reports results using our tool described above which now uses abstract threshold as described in 4.5.1 rather than the concrete one. The results show that our tool with the option abstract threshold (column 2) scales more than the concrete one (column 1) though the number of instances solved are the same. In these benchmarks, though we solve a few more problems than QARMC, it is much faster than our tool.

7. Related Work

The work by Heizmann et al. [7, 34] uses nested word automata to construct a framework for abstraction refinement. Our work could certainly be regarded as extending that framework to tree-structured computations, using tree

[4] http://sv-comp.sosy-lab.org/2015/benchmarks.php
[5] http://clang.llvm.org/

automata instead of (nested) word automata. However our aim is somewhat different. We use automata techniques to *perform* the refinement whereas in [7] automata notation is only used to re-express the verification problem, shifting the verification problem to the construction of "interpolant automata", without providing any automata-based algorithms to do this. On the other hand we discuss the practicality of the automata-based approach on a set of challenging problems.

While we eliminate only one trace at a time in the described procedure, the FTA difference algorithm extends naturally to eliminating (infinite) sets of traces. This is a goal that is well worth pursuing – although to find an interpolant automaton describing an infinite set of infeasible traces is sometimes as difficult as solving the original problem.

Verification of CLP programs using abstract interpretation and specialisation has been studied for some time. The use of an over-approximation of the semantics of a program can be used to establish safety properties – if a state or property does not appear in an over-approximation, it certainly does not appear in the actual program behaviour. A general framework for logic program verification through abstraction was described by Levi [35]. Peralta *et al.* [1] introduced the idea of using a Horn clause representation of imperative languages and a convex polyhedral analyser to discover invariants of a program. Another approach is taken in the work of De Angelis *et al.* [36, 37] on applying program specialisation to achieve verification. Unfolding and folding operations play a vital role in that approach, and hence the program structure is changed much more fundamentally than in our approach.

CEGAR [8] has been successfully used in verification to automatically refine (predicate) abstractions [38, 39] to reduce false alarms but not much has been explored in refining abstractions in the convex polyhedral domain. Some work on this (with progress guarantee) has been done in [40] and [24]. [40] uses the powerset domain, while [24] uses a Hint DAG to gain precision lost during the convex hull operation. Both make use of interpolation. The use of interpolation in refinement in verification of Horn clauses is explored in [41, 42]. In our approach we guarantee elimination of only one trace and elimination of others depends on properties of the abstract interpretation techniques. One drawback of our approach is that we cannot characterize what other infeasible traces are removed by the refinement if there is any. By contrast in interpolation-based techniques the refinement introduces new properties which guarantee progress and the elimination of all counterexamples covered by those properties. However the effectiveness of interpolation-based refinement depends on the generation of "good" interpolants, which is a matter of continuing research, for example by Rümmer *et al.* [31]. There is no generalisation of counterexamples currently since we remove a single counterexample in each iteration. In this sense our refinement approach is weak. The ideas developed in [31] are certainly useful to extend our refinement approach. A number of tools implementing predicate abstraction and refinement are available, such as HSF [29] and BLAST [43]. TRACER [44] is a verification tool based on CLP that uses symbolic execution.

A point of contrast is that in our approach, the refinements are embedded in the clauses whereas in CEGAR they are accumulated in the set of properties used for property-based abstraction. Also we rely on the abstraction using convex polyhedral analysis to discover invariants whereas CEGAR-based approaches rely on interpolation in the refinement stage to perform generalisation, this discovering useful properties. A weakness of invariant generation using interpolation is that the interpolants must share variables with the unsatisfiable part of the constraints, typically those in the integrity constraints, which can be insufficient for finding invariants of inner recursive predicates.Polyhedral analysis is more expensive, yet seems (along with the threshold assertions, see Section 4.5) to be very effective at finding invariants even on the first iteration.

Informally one can say that approaches differ in where the "hard work" is performed. In the CEGAR approaches and in [7] the refinement step is crucial, and interpolation plays a central role. In our approach, by contrast, most of the hard work is done by the abstract interpretation, which finds useful invariants. Finding the most effective balance between abstraction and refinement techniques is a matter of ongoing research.

8. Conclusion and Future work

In this paper we presented a procedure for abstraction refinement in Horn clause verification based on tree automata. This was achieved through a combination of abstraction (using abstraction interpretation) followed by a trace refinement (using finite tree automata). The refinement is independent of the abstract domain used. The practicality of our approach was demonstrated on a set of Horn clause verification problems.

In the future, we will investigate the elimination of a larger set of infeasible traces in each refinement step, possibly by generalising a trace using interpolation or by discovering a set of infeasible traces. At the moment, a new program is generated after refinement and the analysis is restarted from the scratch. In the future, we would like to reuse the result of analysis from the previous iterations and build on this instead of starting the analysis from the scratch. The optimisation of our tool chain is also an important topic for future work as it is clear that our prototype, built by chaining together tools using shell scripts, contains much redundancy.

Acknowledgements

This paper is an extended version of a paper presented at the 16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15). We thank the anonymous referees of VMCAI'15 for useful comments. The paper has been extended by adding proofs of all propositions, adding further information about the implementation and performing some further experiments. We also thank Jorge A. Navas for his help with the tool SeaHorn and for several interesting discussions. B. Kafle was supported by European Commission Framework 7 project ENTRA (Project 318337). J. Gallagher was supported by Danish Research Council grant FNU 10-084290 "Numeric and Symbolic Abstractions for Software Model Checking".

References

[1] J. Peralta, J. P. Gallagher, H. Sağlam, Analysis of imperative programs through analysis of constraint logic programs, in: G. Levi (Ed.), Static Analysis. 5th International Symposium, SAS'98, Pisa, Vol. 1503 of Springer-Verlag Lecture Notes in Computer Science, 1998, pp. 246–261.

[2] S. Grebenshchikov, N. P. Lopes, C. Popeea, A. Rybalchenko, Synthesizing software verifiers from proof rules, in: J. Vitek, H. Lin, F. Tip (Eds.), ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, ACM, 2012, pp. 405–416. doi:10.1145/2254064.2254112.
URL http://doi.acm.org/10.1145/2254064.2254112

[3] G. Delzanno, A. Podelski, Constraint-based deductive model checking, STTT 3 (3) (2001) 250–270.

[4] J. Jaffar, A. E. Santosa, R. Voicu, Modeling systems in CLP, in: M. Gabbrielli, G. Gupta (Eds.), Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings, Vol. 3668 of Lecture Notes in Computer Science, Springer, 2005, pp. 412–413. doi:10.1007/11562931_34.
URL http://dx.doi.org/10.1007/11562931_34

[5] A. Gurfinkel, T. Kahsai, A. Komuravelli, J. A. Navas, The seahorn verification framework. To appear in CAV 2015.

[6] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Semantics-based generation of verification conditions by program specialization, in: M. Falaschi, E. Albert (Eds.), Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015, ACM, 2015, pp. 91–102. doi:10.1145/2790449.2790529.
URL http://doi.acm.org/10.1145/2790449.2790529

[7] M. Heizmann, J. Hoenicke, A. Podelski, Refinement of trace abstraction., in: J. Palsberg, Z. Su (Eds.), Static Analysis, 16th International Symposium, SAS 2009, Vol. 5673 of LNCS, Springer, 2009, pp. 69–85. doi:10.1007/978-3-642-03237-0_7.
URL http://dx.doi.org/10.1007/978-3-642-03237-0_7

[8] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (5) (2003) 752–794.

[9] B. Kafle, J. P. Gallagher, Tree automata-based refinement with application to horn clause verification, in: D. D'Souza, A. Lal, K. G. Larsen (Eds.), Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings, Vol. 8931 of Lecture Notes in Computer Science, Springer, 2015, pp. 209–226. doi:10.1007/978-3-662-46081-8_12.
URL http://dx.doi.org/10.1007/978-3-662-46081-8_12

[10] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, M. Tommasi, Tree automata techniques and applications, Available on: http://www.grappa.univ-lille3.fr/tata, release October, 12th 2007 (2007).

[11] J. P. Gallagher, M. Ajspur, B. Kafle, An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata, Tech. Rep. 145, Roskilde University, Denmark, available from http://akira.ruc.dk/~jpg/dfta.pdf (09 2014).

[12] J. Jaffar, M. Maher, Constraint Logic Programming: A Survey, Journal of Logic Programming 19/20 (1994) 503–581.

[13] J. Jaffar, M. Maher, Constraint Logic Programming: A Survey, in: D. M. Gabbay, C. Hogger, J. Robinson (Eds.), Handbook of Logic in Artificial Intelligence and Logic Programming: Volume 5, Logic Programming, Vol. 5, Oxford University Press, 1998, pp. 591–696.

[14] R. F. Stärk, A direct proof for the completeness of SLD-resolution, in: E. Börger, H. K. Büning, M. M. Richter (Eds.), CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings, Vol. 440 of LNCS, Springer, 1989, pp. 382–383.

[15] J. P. Gallagher, L. Lafave, Regular approximation of computation paths in logic and functional languages, in: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation, Vol. 1110 of Springer-Verlag Lecture Notes in Computer Science, 1996, pp. 115–136.

[16] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: R. M. Graham, M. A. Harrison, R. Sethi (Eds.), POPL, ACM, 1977, pp. 238–252.

[17] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, 1978, pp. 84–96.

[18] F. Benoy, A. King, Inferring argument size relationships with CLP(R), in: J. P. Gallagher (Ed.), Logic-Based Program Synthesis and Transformation (LOPSTR'96), Vol. 1207 of LNCS, 1996, pp. 204–223.
[19] N. Halbwachs, Y. E. Proy, P. Raymound, Verification of linear hybrid systems by means of convex approximations, in: Proceedings of the First Symposium on Static Analysis, Vol. 864 of LNCS, 1994, pp. 223–237.
[20] R. Bagnara, P. M. Hill, E. Zaffanella, The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems, Science of Computer Programming 72 (1–2) (2008) 3–21.
[21] L. Lakhdar-Chaouch, B. Jeannet, A. Girault, Widening with thresholds for programs with complex control graphs, in: T. Bultan, P.-A. Hsiung (Eds.), ATVA 2011, Vol. 6996 of LNCS, Springer, 2011, pp. 492–502.
[22] B. Kafle, J. P. Gallagher, Constraint specialisation in horn clause verification, in: K. Asai, K. Sagonas (Eds.), Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015, ACM, 2015, pp. 85–90. doi:10.1145/2678015.2682544.
URL http://doi.acm.org/10.1145/2678015.2682544
[23] J. P. Gallagher, B. Kafle, Analysis and transformation tools for constrained Horn clause verification, TPLP 14 (4-5 (additional materials in online edition)) (2014) 90–101.
URL http://journals.cambridge.org/action/displaySuppMaterial?cupCode=1&type=4&jid=TLP&volumeId=14&issueId=4-5&aid=9303163
[24] B. S. Gulavani, S. Chakraborty, A. V. Nori, S. K. Rajamani, Automatically refining abstract interpretations, in: C. R. Ramakrishnan, J. Rehof (Eds.), TACAS, Vol. 4963 of LNCS, Springer, 2008, pp. 443–458.
[25] J. Jaffar, V. Murali, J. A. Navas, A. E. Santosa, Tracer: A symbolic execution tool for verification, in: P. Madhusudan, S. A. Seshia (Eds.), CAV, Vol. 7358 of LNCS, Springer, 2012, pp. 758–766.
[26] A. Gupta, A. Rybalchenko, Invgen: An efficient invariant generator, in: A. Bouajjani, O. Maler (Eds.), CAV, Vol. 5643 of LNCS, Springer, 2009, pp. 634–640.
[27] D. Beyer, Second competition on software verification - (summary of sv-comp 2013), in: N. Piterman, S. A. Smolka (Eds.), TACAS, Vol. 7795 of LNCS, Springer, 2013, pp. 594–609.
[28] A. Podelski, A. Rybalchenko, ARMC: the logical choice for software model checking with abstraction refinement., in: M. Hanus (Ed.), PADL 2007, Vol. 4354 of LNCS, 2007, pp. 245–259.
[29] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, A. Rybalchenko, Hsf(c): A software verifier based on Horn clauses - (competition contribution), in: C. Flanagan, B. König (Eds.), TACAS, Vol. 7214 of LNCS, Springer, 2012, pp. 549–551.
[30] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Program verification via iterated specialization, Sci. Comput. Program. 95 (2014) 149–175. doi:10.1016/j.scico.2014.05.017.
URL http://dx.doi.org/10.1016/j.scico.2014.05.017
[31] P. Rümmer, H. Hojjat, V. Kuncak, Disjunctive interpolants for horn-clause verification, in: N. Sharygina, H. Veith (Eds.), CAV 2013, Proceedings, Vol. 8044 of LNCS, Springer, 2013, pp. 347–363.
[32] D. Beyer, Software verification and verifiable witnesses - (report on SV-COMP 2015), in: Baier and Tinelli [45], pp. 401–416. doi:10.1007/978-3-662-46681-0_31.
URL http://dx.doi.org/10.1007/978-3-662-46681-0_31
[33] A. Gurfinkel, T. Kahsai, J. A. Navas, Seahorn: A framework for verifying C programs (competition contribution), in: Baier and Tinelli [45], pp. 447–450. doi:10.1007/978-3-662-46681-0_41.
URL http://dx.doi.org/10.1007/978-3-662-46681-0_41
[34] M. Heizmann, J. Hoenicke, A. Podelski, Nested interpolants, in: M. V. Hermenegildo, J. Palsberg (Eds.), Proceedings of POPL 2010, ACM, 2010, pp. 471–482.
[35] G. Levi, Abstract interpretation based verification of logic programs, Electr. Notes Theor. Comput. Sci. 40 (2000) 243.
[36] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Verifying programs via iterated specialization, in: E. Albert, S.-C. Mu (Eds.), PEPM, ACM, 2013, pp. 43–52.
[37] E. De Angelis, F. Fioravanti, A. Pettorossi, M. Proietti, Verimap: A tool for verifying programs through transformations, in: E. Ábrahám, K. Havelund (Eds.), TACAS, Vol. 8413 of LNCS, Springer, 2014, pp. 568–574.
[38] M. Burke, M. L. Soffa (Eds.), Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001, ACM, 2001.
[39] J. Launchbury, J. C. Mitchell (Eds.), Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002, ACM, 2002.
[40] A. Albarghouthi, A. Gurfinkel, M. Chechik, Craig interpretation, in: A. Miné, D. Schmidt (Eds.), SAS, Vol. 7460 of LNCS, Springer, 2012, pp. 300–316.
[41] N. Bjørner, K. L. McMillan, A. Rybalchenko, On solving universally quantified Horn clauses, in: F. Logozzo, M. Fähndrich (Eds.), SAS, Vol. 7935 of LNCS, Springer, 2013, pp. 105–125.
[42] A. Gupta, C. Popeea, A. Rybalchenko, Solving recursion-free Horn clauses over li+uif, in: H. Yang (Ed.), APLAS, Vol. 7078 of LNCS, Springer, 2011, pp. 188–203.
[43] T. Ball, V. Levin, S. K. Rajamani, A decade of software model checking with SLAM, Commun. ACM 54 (7) (2011) 68–76.
[44] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, P. J. Stuckey, Failure tabled constraint logic programming by interpolation, TPLP 13 (4-5) (2013) 593–607.
[45] C. Baier, C. Tinelli (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, Vol. 9035 of Lecture Notes in Computer Science, Springer, 2015. doi:10.1007/978-3-662-46681-0.
URL http://dx.doi.org/10.1007/978-3-662-46681-0