Tree automata-based refinement with application to Horn clause verification

Kafle, Bishoksan; Gallagher, John Patrick

Published in:
Verification, Model Checking, and Abstract Interpretation

DOI:
10.1007/978-3-662-46081-8_12

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Kafle, B., & Gallagher, J. P. (2015). 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* (Vol. 8931, pp. 209-226). Springer. Lecture Notes in Computer Science No. 8931 https://doi.org/10.1007/978-3-662-46081-8_12

Tree Automata-Based Refinement with Application to Horn Clause Verification

Bishoksan Kafle[1,⋆] and John P. Gallagher[1,2,⋆⋆]

[1] Roskilde University, Denmark
[2] 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.

1 Introduction

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 [23]; 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.

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.

⋆ Supported by EU FP7 project ENTRA (Project 318337).
⋆⋆ Supported by Danish Research Council grant FNU 10-084290.

D. D'Souza et al. (Eds.): VMCAI 2015, LNCS 8931, pp. 209–226, 2015.

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, leading to more precise abstractions which can lead to better invariant generation in subsequent iterations. The refinement is manifested in the refined clauses, rather than in an accumulated set of properties as in the counterexample-guided abstraction refinement (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. 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 3). (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 3.5) (3) We propose a "splitting" operator on FTAs (Section 2) and describe its role in Horn clause verification (Section 4.1). (4) We demonstrate the feasibility of our approach in practice applying it to Horn clause verification problems (Section 5).

2 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, $Q \subseteq Q_f$ 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 $t \Rightarrow^* q$.

2.1 Operations on FTAs

FTAs are closed under Boolean set operations, but for our purposes we mention only union and difference 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 $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^d)$ [9]. 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 $\mathcal{A}$ [9] is not viewed as a practical procedure for manipulating tree automata, even fairly small ones. In a recent work Gallagher *et al.* [14] 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 [14].

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

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.*

We have $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.

3 Horn Clauses and Their Trace Automata

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

3.1 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 $Z_1, \ldots, Z_n$. 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 each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator (called S_P^D in [25, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D. Furthermore $\mathsf{lfp}(S_P^D)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. 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.

3.2 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 satisifying P also satisfies F.

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

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. Due to the equivalence of the minimal model of P with the set of atomic logical consequences of P, we have yet another equivalent formulation of the CHC verification problem.

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

```
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).
```

Fig. 1. Example CHCs. The McCarthy 91-function.

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.

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

The elements of $\mathcal{L}(\mathcal{A}_P)$ are called trace terms for P. In Section 4 we will see that several clauses differing only in their predicate names are assigned the same function symbol.

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.

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:

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

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 [33] and [16].

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 renamed variant of) some clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ such that $c_j = \mathsf{id}_P(p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k))$; 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 $\mathtt{node_i}$ represents a label in the trace. Then $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ is defined as:

$$\begin{array}{ll} Q & = \{\mathtt{node_1}, \mathtt{node_2}, \mathtt{node_3}, \mathtt{node_4}\} \\ Q_f & = \{\mathtt{node_1}\} \\ \Sigma & = \{c_1, c_2, c_3, c_4\} \\ \Delta & = \{c_1 \to \mathtt{node_3},\ c_1 \to \mathtt{node_4},\ c_2(\mathtt{node_3}, \mathtt{node_4}) \to \mathtt{node_2}, \\ & \quad\ c_3(\mathtt{node_2}) \to \mathtt{node_1}\} \end{array}$$

and Σ is the same as in $\mathcal{A}_P$. The trace t is not feasible since $\mathsf{constr}(t) = \{\mathtt{A} \leq 100, \mathtt{B} > 91, \mathtt{A} \leq 100, \mathtt{C} = \mathtt{A} + 11, \mathtt{C} > 100, \mathtt{D} = \mathtt{C} - 10, \mathtt{D} > 100, \mathtt{B} = \mathtt{D} - 10\}$ and this is not satisfiable.

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 labelling 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 a standard result from constraint logic programming [25] in terms of the concepts defined above.

Proposition 1. *Let P be a set of CHCs.*

1. *Then for all $t \in \mathcal{L}(\mathcal{A}_P)$ the constrained trace atom for t is a logical consequence of P. (Note that if t is not feasible this is trivially true).*

2. *If $p(a)$ is in the minimal model of P, 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 $\phi[X/a]$ is true.*

Assuming that the constraint theory has a complete satisfiability procedure, part 1 of Proposition 1 corresponds to the standard soundness result for resolution-based proof systems, and part 2 corresponds to completeness.

3.4 Model-Preserving Transformation of Trace Automata

Proposition 1 implies that the constrained trace atoms for the feasible traces describe exactly the elements of the minimal model, which is equivalent to the set of atomic logical consequences of P. As a consequence the set of feasible traces in $\mathcal{L}(\mathcal{A}_P)$ can be regarded as a representation of the minimal model of P.

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 4).

3.5 Generation of CHCs from a Trace FTA

Now we describe a procedure (Algorithm 1) 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 but we omit that here. 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 1 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 1 is expressed by the following proposition.

Proposition 2. *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 1. Then $\mathcal{L}(\mathcal{A}_{P'}) = \mathcal{L}(\mathcal{A})$. Furthermore if $\mathcal{L}(\mathcal{A}_{P'})$ 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.*

Input: An FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ and a set of Horn clauses P
Output: A set of Horn clauses P'
$P' \leftarrow \emptyset$;
for *each* $c_i(q_1, \ldots, q_n) \rightarrow q$ *(where* $n \geq 0$*)* $\in \Delta$ **do**
  let $c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$ be any 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 1. ALGORITHM for generating a set of clauses from an FTA

Example 3 (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`}, 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 1 are the following, with the renaming function $\rho = \{$`[false]` $\mapsto$ `false`, `[mc91]` $\mapsto$ `mc91`, `[e,false]` $\mapsto$ `false_1`, `[mc91,e1]` $\mapsto$ `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).
```

3.6 Abstract Interpretation of Constrained Horn Clauses

Abstract interpretation [10] is a static program analysis techniques which derives sound over-approximations by computing abstract fixed points. Convex polyhedron analysis (CPA) [11] is a program analysis technique based on abstract interpretation [10]. 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 [4].

We summarise briefly the elements of convex polyhedron analysis for CHC; further details (with application to CHC) can be found in [11,4]. 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 [11] 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. Much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [22]. 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 [2] to implement the operations on convex polyhedra, and incorporates a threshold generation phase based on the method described by Lakhdar-Chaouch *et al.* [27], as well as a constraint strengthening pre-processing which propagates constraints both forwards and backwards in the clauses of P. Space does not permit a detailed explanation.

4 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 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 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 2.

Input: A set of Horn clauses P and an infeasible trace t
Output: A set of 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 1) ;

return P';

Algorithm 2. ALGORITHM for clause refinement

Both $\mathcal{A}_P$ and $\mathcal{A}_t$ in Algorithm 2 are deterministic 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, the restructuring of the clauses required to remove a trace can split the predicates. This restructuring is the 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 2. In particular $\mathsf{false} \in M[\![P]\!]$ if and only if $\mathsf{false} \in M[\![P']\!]$ (assuming that the predicate renaming at least preserves the predicate name false).

Example 4. Consider again the FTA shown in Example 3. 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, using the same renaming function as in Example 3. 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].
```

```
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).
```

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 3 (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'.*

After the removal of the trace t (step 3 of Algorithm 2) the language of $\mathcal{A}_P \setminus \mathcal{A}_t$ does not contain t. Then using Algorithm 1 to generate P', t will not be a possible trace in P'. It is physically impossible to construct t, in any abstract domain.

4.1 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 [15].

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.

5 Experiments on CHC Benchmark Problems

Our tool consists of an implementation of a *convex polyhedra analyser* for CLP written in Ciao Prolog[1] interfaced to the Parma Polyhedra Library [2] as well as an implementation of an FTA determiniser written in Java. It takes as input a

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

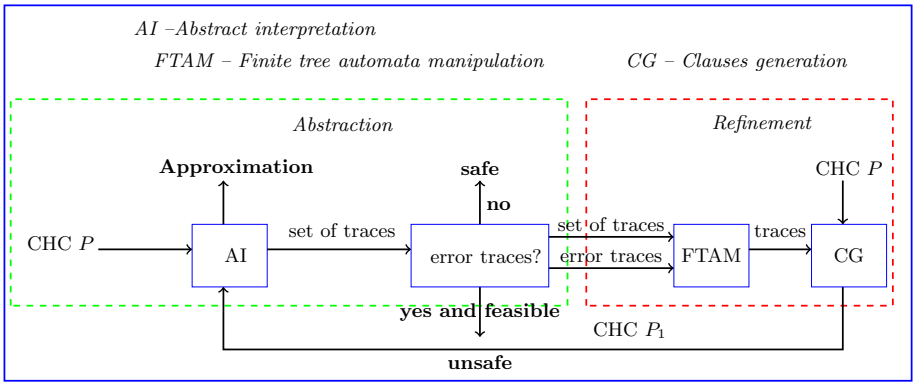

Fig. 2. *Abstraction-refinement scheme in Horn clause verification*

CLP program and returns "safe", "unsafe" or "unknown" (after timeout). 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 [19] (21 problems), TRACER [26] (66 problems), InvGen [21] (68 problems), and also from the TACAS 2013 Software Verification Competition [5] (52 problems). Most of these problems are available in `C` 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 are available from `http://akira.ruc.dk/~kafle/VMCAI15-Benchmarks.zip`. The experiments were carried out on an Intel(R) quad-core computer with a 2.66GHz processor running Debian 5 in 6 GB memory.

5.1 Summary of Results

The results of our experiments are summarised in Table 3. Column CPA summarises the results using our own *convex polyhedra analyser* (Section 3.6) with no refinement step. Column CPA+R shows the results obtained by iterating the CPA algorithm with the refinement step described in Section 4, Algorithm 2. Column CPA+R+Split incorporates the FTA-based state splitting into the refinement step (Section 4.1). Column QARMC shows the results obtained on the same problems using the QARMC tool [31].

5.2 Discussion of Results

The results show that CPA is reasonably effective on its own, solving 74% (160/216) of the problems, though it times out for seven problems. When combined with a refinement phase we can solve 22 further problems. Although only

[2] Thanks to Emanuele De Angelis for the translation.

	CPA	CPA+R	CPA+R+Split	QARMC
solved (safe/unsafe)	160 (142/18)	182 (160/22)	195 (164/31)	178 (141/37)
unknown/ timeout	49/7	-/34	-/22	-/38
average time (secs.)	5.98	51.66	50.08	59.1
% solved	74	84.25	90.27	82.4

Fig. 3. Experimental results on 216 (179 safe / 37 unsafe) CHC verification problems with a timeout of five minutes

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 [15]. 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. QARMC tends to perform more (but faster) iterations.

Our implementation uses the product form for DFTAs produced by the determinisation algorithm, although the formalisation of refinement in Section 4 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.

5.3 Comparison with Other Tools

Our results improve on QARMC both in average time and the number of instances solved. 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. Convex polyhedral analysis is good at finding the required invariants to prove a program safe and due to this we solved more safe problems than QARMC. QARMC seems to be more effective at finding bugs. Most of the problems challenging to us come from particular categories e.g. SystemC (modelled over fixed size integers) and Control Flow and Integer Variables of [5] 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 Related Work

The work by Heizmann et al. [23,24] uses 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 automata instead of (nested) word automata. However our aim is rather different. We use automata techniques to *perform* the refinement whereas in [23] 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. However in our setting that does not seem a useful goal – to find an automaton describing an infinite set of infeasible traces often amounts to 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 [29]. Peralta *et al.* [30] 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.* [12,13] 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 [7,28] 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 [1] and [19]. [1] uses the powerset domain, while [19] 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 [6,20]. In our approach we guarantee elimination of only one trace and elimination of others depends on properties of the abstract interpretation techniques. 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.* [32]. A number of tools implementing predicate abstraction and refinement are available, such as HSF [18] and BLAST [3]. TRACER [17] 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 discover relevant

properties. Polyhedral analysis is more expensive, yet seems (along with the threshold assertions, see Section 3.6) to be very effective at finding invariants even on the first iteration. 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. Informally one can say that approaches differ in where the "hard work" is performed. In the CEGAR approaches and in [23] 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.

7 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. 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. We thank the anonymous referees for useful comments.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)
2. Bagnara, R., Hill, P.M., Zaffanella, E.: 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), 3–21 (2008)
3. Ball, T., Levin, V., Rajamani, S.K.: A decade of software model checking with SLAM. Commun. ACM 54(7), 68–76 (2011)
4. Benoy, F., King, A.: Inferring argument size relationships with CLP($\mathcal{R}$). In: Gallagher, J. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
5. Beyer, D.: Second competition on software verification - (summary of SV-COMP 2013). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)

6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (eds.) SAS 2013. LNCS, vol. 7935, pp. 105–125. Springer, Heidelberg (2013)
7. Burke, M., Soffa, M.L. (eds.): Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22. ACM (2001)
8. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
9. Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications (2007), `http://www.grappa.univ-lille3.fr/tata` (release October 12, 2007)
10. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL, pp. 238–252. ACM (1977)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, pp. 84–96 (1978)
12. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying programs via iterated specialization. In: Albert, E., Mu, S.-C. (eds.) PEPM, pp. 43–52. ACM (2013)
13. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: VeriMAP: A tool for verifying programs through transformations. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 568–574. Springer, Heidelberg (2014)
14. Gallagher, J.P., Ajspur, M., Kafle, B.: An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata. Technical Report 145, Roskilde University, Denmark, (September 2014), `http://akira.ruc.dk/~jpg/dfta.pdf`
15. Gallagher, J.P., Kafle, B.: Analysis and transformation tools for constrained Horn clause verification. TPLP, 14(4-5) (additional materials in online edition), 90–101 (2014)
16. Gallagher, J.P., Lafave, L.: Regular approximation of computation paths in logic and functional languages. In: Danvy, O., Thiemann, P., Glück, R. (eds.) Dagstuhl Seminar 1996. LNCS, vol. 1110, pp. 115–136. Springer, Heidelberg (1996)
17. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Failure tabled constraint logic programming by interpolation. TPLP 13(4-5), 593–607 (2013)
18. Grebenshchikov, S., Gupta, A., Lopes, N.P., Popeea, C., Rybalchenko, A.: HSF(C): A software verifier based on Horn clauses - (competition contribution). In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 549–551. Springer, Heidelberg (2012)
19. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
20. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 188–203. Springer, Heidelberg (2011)
21. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
22. Halbwachs, N., Proy, Y.E., Raymound, P.: Verification of linear hybrid systems by means of convex approximations. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994)

23. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 69–85. Springer, Heidelberg (2009)
24. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of POPL 2010, pp. 471–482. ACM (2010)
25. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19/20, 503–581 (1994)
26. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: A symbolic execution tool for verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 758–766. Springer, Heidelberg (2012)
27. Lakhdar-Chaouch, L., Jeannet, B., Girault, A.: Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 492–502. Springer, Heidelberg (2011)
28. Launchbury, J., Mitchell, J.C. (eds.): Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18. ACM (2002)
29. Levi, G.: Abstract interpretation based verification of logic programs. Electr. Notes Theor. Comput. Sci. 40, 243 (2000)
30. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)
31. Podelski, A., Rybalchenko, A.: ARMC: The logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
32. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013)
33. Stärk, R.F.: A direct proof for the completeness of SLD-resolution. In: Börger, E., Kleine Büning, H., Richter, M.M. (eds.) CSL 1989. LNCS, vol. 440, pp. 382–383. Springer, Heidelberg (1990)