Constraint specialisation in Horn clause verification

Bishoksan Kafle[*]

Roskilde University, Denmark

John P. Gallagher[*]

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

Abstract

We present a method for specialising the constraints in constrained Horn clauses with respect to a goal. We use abstract interpretation to compute a model of a query-answer transformed program of a given set of clauses and a goal. The effect is to propagate the constraints from the goal top-down and propagate answer constraints bottom-up. We use the constraints from the model to compute a specialised version of each clause in the program. The specialisation procedure can be repeated to yield further specialisation. The approach is independent of the abstract domain and the constraint theory underlying the clauses. Experimental results on verification problems show that this is an effective transformation, both in our own verification tools (convex polyhedra analyser) and as a pre-processor to other Horn clause verification tools.

Keywords: constraint specialisation, query-answer transformation, Horn clauses, abstract interpretation, convex polyhedral analysis

1. Introduction

In this paper, we present a method for specialising the constraints in constrained Horn clauses, CHCs in short (also called constraint logic programs) with respect to a goal. The verification problem that we address has the following setting: a set of constrained Horn clauses formalises some system and the goal is an atomic formula representing a property of that system. We wish to check whether the goal is a consequence of the Horn clauses. The constraint specialisation procedure uses abstract interpretation to compute constraints propagated both from the goal top-down and constraints in the clauses bottom-up. Then we construct a specialised version of each clause by inserting the relevant constraints, without unfolding the clauses at all. As a result, each clause is further strengthened or removed altogether, while preserving the derivability of the goal.

Verification of this specialised set of clauses becomes more effective since some implicit invariants in the original clauses are discovered and made explicit in the specialised version. A central problem in all automatic verification procedures is to find invariants, and this is the

[*]Corresponding author

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

```
int a ,b;
while (*) { // loop invariant l(a,b)
        a = a + b;
        b = b + 1;
}
```

(a) Example program Loop_add

$$a = 1 \land b = 0 \rightarrow l(a,b)$$
$$l(a',b') \land a = a' + b' \land b = b' + 1 \rightarrow l(a,b)$$
$$l(a,b) \rightarrow a \geq b$$

(b) Verification conditions for Loop_add

Figure 1: Motivating example

```
c1. l(A,B):- A=1, B=0.
c2. l(A,B):- A=C+D, B=D+1, l(C,D).
c3. false :- B>A, l(A,B).
```

Figure 2: Verification conditions for Loop_add in CLP syntax

underlying reason for the usefulness of our constraint specialisation procedure. The approach is independent of the abstract domain and the constraint theory.

While specialisation has been applied to verification and analysis problems before, the novelty of our procedure is to do specialisation without any unfolding. The only specialisation is to strengthen constraints within each clause, possibly eliminating a clause if its constraints become unsatisfiable. This seems to capture the essence of the role of constraint propagation, separated from other operations such as clause unfolding. Somewhat surprisingly, this apparently limited form of specialisation is capable of handling a lot of verification benchmarks on its own; on the other hand, due to its simple form, constraint specialisation is a useful pre-processing step for verification tools incorporating a wider range of techniques. We postulate that making invariants explicit contributes positively to the effect of other constraint manipulation operations such as widening and interpolation. We therefore present the procedure as a useful tool in a toolbox for verification of constrained Horn clauses.

Motivating example. We present an example program in Figure 1. The problem is to show that if $a = 1 \land b = 0$ holds before executing the program Loop_add in Figure 1a (taken from [16]) then $a \geq b$ holds after executing it. The problem is considered safe if the Hoare triple $\{a = 1 \land b = 0\}$ Loop_add $\{a \geq b\}$ is valid. Figure 1b shows the Horn clauses whose satisfiability establishes this property (its equivalent representation in Constraint Logic Programing (CLP) syntax is shown in Figure 2). Please note that the last clause in Figure 1b can be equivalently written as $l(a,b) \land b > a \rightarrow false$. It is a simple but still challenging problem for many verification tools for constrained Horn clauses. The invariant $a \geq b \land b \geq 0$ on the predicate $l(a,b)$ proves this program safe. Finding this invariant is a challenging task for many state of the art verification tools. For example QARMC [30] or SeaHorn [33] (using only the PDR engine) do not terminate on this program. However, SeaHorn (with PDR and the abstract interpreter IKOS) solves it in less than a second. The tool based on specialisation of Horn clauses [16] needs at least forward and backward iteration to solve this problem. We discuss how our constraint specialisation solves this example without needing further iteration.

1.1. Related Work

There is a good deal of related work, since our procedure draws on a number of different techniques which have been applied in different contexts and languages. The basic specialisation and analysis techniques that we apply are well known, though we are not aware of previous work combining them in the way we did or applying them effectively in verification problems.

Constraint specialisation. Methods for strengthening the constraints in logic programs go back at least to the work of Marriott *et al.* on most specific logic programs [59]. In that work the constraints were just equalities between terms and the strengthening was goal-independent. We say a clause $H \leftarrow \phi, B_1, \ldots, B_n$ is strengthened version of $H \leftarrow \psi, B_1, \ldots, B_n$ if $\phi \rightarrow \psi$. This can be extened to a (logic) program if all clauses in the program are strengthened. In [24] the idea was similar but it was extended to strengthen constraints while preserving the answers with respect to a goal. Constraint strengthening was also applied for extracting determinacy in logic program executions [14], in a goal-dependent setting, and arithmetic constraints were also handled. The purpose of constraint strengthening in these works was to improve execution efficiency, for example by detecting failure earlier or to allow more efficient compilation.

Furthermore the idea of constraint propagation leading to specialisations is widespread in constraint-based languages and problem-solving frameworks [60, 70] as well as in partial evaluation and program specialisation [22, 23, 25, 42, 44, 51, 52, 54, 55, 72].

Query-answer transformations and related methods. A central tool in our procedure is query-answer transformation, which is related to the so-called "magic set" transformation. We prefer the terminology query-answer transformation to the name magic set transformation in the literature, as it reflects the purpose of the transformation. The verification problem that we consider in this paper is whether a given goal A is a consequence of a set of Horn clauses P. The relevance of the query-answer transformation to this problem is that the search for a proof can be attempted "top-down" starting from the error state encoded by the predicate A, or "bottom-up" starting from the program's behaviour characterised by P. A query-answer transformation specialises the clauses in P with respect to the goal A and allows bottom-up and top-down search to be combined.

The "magic set" transformation originated in the field of Datalog query processing in the 1980s [3, 68, 74]. In the Datalog context, the transformation was invented to combine the efficiency of bottom-up evaluation with the focussed evaluation of a top-down search starting from a given query. The transformation with respect to the query returns a set of Datalog rules and facts that are extensions of the original ones except that they contain extra conditions (the so-called "magic predicates") expressing the dependence on the query, together with extra rules for these magic predicates. The resulting rules and facts can be evaluated "bottom-up" allowing efficient algorithms for bottom-up evaluation to be applied, without the potential inefficiency of an undirected goal-independent search. The "magic sets" and "magic templates" techniques for Datalog also incorporate input-output modes, but the more general notion of query-answer transformation that we use does not require these.

Applications of query-answer transformations in logic program analysis [10, 14, 20, 27] have a similar motivation. Analysis of a program can be either goal-independent, deriving an approximation of the model of a program which implicitly expresses the behaviour of all goals, or goal-dependent, deriving an approximation of a top-down derivation of a specific goal. Algorithms for goal-independent analysis are generally simpler to implement, but precision of an analysis can be lost compared to a goal-dependent analysis. A query-answer transformations allows the goal-dependence to be "compiled in" to the program clauses; analysis of the resulting transformed clauses using a goal-independent analysis framework yields results that are at least as precise as with a goal-dependent analysis.

Although not formulated as a program transformation, the semantic concept of a *minimal function graph* is related to query-answer transformations. Given a function $f : A \rightarrow B$, its function graph is the set of pairs $\{x \mapsto f(x) \mid x \in A\}$. Given some "call" to the function, say $f(a)$, the minimal function graph is the smallest set of pairs $x \mapsto f(x)$ sufficient to evaluate $f(a)$ (ignoring calls to subsidiary functions). This is in general a subset of the function graph of f. Minimal function graph semantics have been formulated for both functional [43, 45] and logic programming [25, 75] languages and applied to program analysis problems. The "query" or "magic" predicates of the transformations appear in minimal function graph constructions as a set of function invocations computed top-down from the call. Informally, a query-answer transformation for logic programs could be viewed as a "compilation" of the minimal graph semantics with respect to a specific program and goal, though further study is needed to formalise this view.

Abstract interpretation. Abstract interpretation [11] is a static program analysis technique which derives sound over-approximations of programs by computing abstract semantics. Abstract interpretation over a domain of convex polyhedra was first achieved by Cousot and Halbwachs [13] and applied to constraint logic programs by Benoy and King [4]. Abstract interpretation over convex polyhedra was incorporated in a program specialisation algorithm by Peralta and Gallagher [65]. The method of widening with thresholds for increasing the precision of widening over the domain of convex polyhedra was first presented by Halbwachs *et al.* [35]. We adapted and applied a technique for generating threshold constraints presented by Lakhdar-Chaouch *et al.* [53]. We apply abstract interpretation over the domain of convex polyhedra to derive an over-approximation of the model of the query-answer transformed program.

Verification by specialisation. The use of program transformation to verify properties expressed as constraint logic programs was pioneered by Pettorossi and Proietti [66] and Leuschel [56] and continues in recent work by De Angelis *et al.* [15, 17]. Transformations that preserve the minimal model (or other suitable semantics) of logic programs are applied systematically to make properties explicit. Our tool can be regarded as identifying the essence of these tools, namely constraint propagation forwards and backwards within the program. The program specialisation technique supercompilation [72], which also inherently involves constraint propagation through "driving", was applied as a tool to verify statements [49, 50, 57]. Supercompilation is a language independent concept originally developed for the functional language REFAL and later adapted for some other languages as well.

CLP verification tools. Verification of CLP programs has been studied for some time. Our aim in this paper is not to demonstrate a new verification tool but to identify a transformation that can often verify programs on its own and also benefits CLP verification tools generally as a pre-processor. The work closest to ours is by De Angelis et al. [16]; that method also includes forward and backward propagation of constraints using *fold-unfold* transformation. Furthermore the forward propagation method in that work uses a program reversal which can only be applied to linear Horn clauses (a transition system) whereas we can handle also non-linear clauses. However there are various methods for linearisation of Horn clauses in the literature [18, 47]. The resulting program in their approach can blow up in size with respect to the original program, whereas constraint specialisation cannot.

Much other work on CLP verification exists, much of it based on property abstraction and refinement using interpolation, for example [9, 29, 67, 2, 8, 31]. Our specialisation technique is not directly comparable to these methods, but as we have shown in experiments with QARMC and Eldarica, constraint specialisation can be used as a pre-processor to such tools, increasing their effectiveness. The model checking algorithm implemented in Eldarica for Horn clause verification is similar in spirit to the one described in [29] but uses disjunctive interpolation for counterexamples generalisation, instead of tree interpolation, which is strictly more general than tree interpolation [69]. The approach described in this paper is used as a pre-processor of Horn clauses in the tool RAHFT [48].

1.2. Overview and contributions of the paper

In Section 2 the relevant notation and theory concerning CHCs is presented. Following this, Section 3 describes various methods and techniques employed in our procedure, in the form required. These include an algorithm for computing an abstract interpretation of a set of CHCs over the domain of convex polyhedra in Section 3.1 and the details of the query-answer transformation in Section 3.2. Section 4 contains the main contribution, namely the procedure for constraint specialisation. Section 5 puts the procedure in the context of verification, explaining the role of CHC integrity constraints. Section 6 contains the experimental evaluation of the procedure. Finally Section 7 presents the conclusions.

The contributions of this work are as follows.

- We present a method for specialising the constraints in the clauses using query-answer transformation and abstract interpretation (see Section 4).
- We demonstrate the effectiveness of the transformation by applying it to Horn clause verification problems (see Section 6).

Experimental results on verification problems show that this is an effective transformation, propagating information both backwards from the statement to be proved, and forwards from the Horn clauses. We show its effectiveness both in our own verification tools and as a pre-processor to other Horn Clause verification tools. In particular, we run our specialisation procedure as a pre-processor to our *convex polyhedra analyser*, to the state of the art verification tools like QARMC [29, 67] and Eldarica [37].

2. Preliminaries

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geq 0)$, 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. The arguments of a predicate are always regarded as a tuple; when we write $p(X)$ or $p(a)$, then X and a stand for (possibly empty) tuples of variables and constants respectively.

A set of CHCs can be regarded as a CLP program. Unlike CLP, CHCs are not always regarded as executable programs, but rather as specifications or semantic representations of other formalisms. However the semantic equivalence of CHC and CLP allows that techniques developed in one framework are applicable to the other. We follow the syntactic conventions of CLP and write a Horn clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$. In this paper we take the constraint theory to be linear arithmetic with the relation symbols $\leq, \geq, <, >$ and $=$, but the contributions of the paper are independent of the constraint theory.

2.1. Interpretations and models

An interpretation of a set of CHCs is a truth assignment to each atomic formula $p(a)$, where p is a predicate and a is a tuple of constants from the constraint theory. An interpretation is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z)$ where Z is a tuple of distinct variables and ϕ is a constraint over Z. The constrained fact $A \leftarrow \phi$ stands for the set of ground facts $A\theta$ (where θ is a grounding substitution) such that $\phi\theta$ holds in the constraint theory. An interpretation M is the set of all ground facts denoted by its elements. $M_1 \subseteq M_2$ if the set of denoted ground facts of M_1 is contained in the set of denoted ground facts of M_2. An interpretation M satisfies a CHC $p_0(X_0) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$, if M contains constrained facts $\{p_i(X_i) \leftarrow \phi_i \mid 0 \leq i \leq n\}$, and $\forall(\phi_0 \leftarrow (\phi \wedge \bigwedge_{i_1}^{n} \phi_i))$ is true.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. A set of CHCs P has a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$. Let S_P^D be the immediate consequences operator, an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D [38, Section 4]. $M[\![P]\!]$, which is equal to the least fixed point of S_P^D, can be computed as the limit of the sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. The abstract interpretation of CHC clauses presented in Section 3.1, uses this sequence as the basis of the model computation. From now on, whenever we talk about a model of Horn clauses, we refer to its minimal model.

Given two constraints ϕ and ψ over some constraint theory T, we say ϕ is stronger than ψ if $T \models \forall(\phi \rightarrow \psi)$.

3. Methods and techniques

This section describes the techniques used in the constraint specialisation procedure, including abstract interpretation over convex polyhedra and the query-answer transformation. These are methods drawn from the literature; the contribution of the section is to present them in a form suitable for integration in the procedure, and present an efficient implementation of the abstract interpretation of CHCs.

3.1. Abstract Interpretation over the domain of convex polyhedra

Convex polyhedral analysis (CPA) [13] is a program analysis technique based on abstract interpretation [11]. 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 CPA to CHCs was by Benoy and King [4]. In this section we develop an algorithm for CPA of CHCs incorporating a number of features enhancing precision and efficiency.

We summarise briefly the elements of convex polyhedral analysis for CHC; further details can be found in [13, 4]. Let $\mathcal{A}^k$ be the set of convex polyhedra of dimension k. Let P be a set of CHCs containing n predicates, say $p_1, \ldots, p_n$, where the arity of p_i is $\mathsf{ar}(p_i)$. The *abstract domain* $\mathcal{D}_P$ for P (or just $\mathcal{D}$ when P is clear from context) is the set of n-tuples of convex polyhedra of the respective dimension, that is $\mathcal{D} = \mathcal{A}^{\mathsf{ar}(p_1)} \times \cdots \times \mathcal{A}^{\mathsf{ar}(p_n)}$. Let the empty polyhedron of dimension k be denoted $\bot_k$ (or just $\bot$ when the dimension is clear from context). Inclusion of polyhedra is a partial order on $\mathcal{A}^k$ and the partial order $\sqsubseteq$ on $\mathcal{D}$ is its point-wise extension. The convex hull of two polyhedra $d_1, d_2 \in \mathcal{A}^k$ is denoted $d_1 \sqcup d_2$, and the least upper bound $\sqcup$ of tuples in $\mathcal{D}_P$, say $\langle d_1, \ldots, d_n \rangle$ and $\langle e_1, \ldots, e_n \rangle$, is $\langle d_1 \sqcup e_1, \ldots, d_n \sqcup e_n \rangle$. Given an element $\langle d_1,\ldots,d_n\rangle \in \mathcal{D}_P$, define the *concretisation* function γ such that $\gamma(\langle d_1,\ldots,d_n\rangle) = \{\langle p_1(a_1),\ldots,p_n(a_n)\rangle \mid a_i \text{ is a point in } d_i, 1 \le i \le n\}$. Let an *abstract semantic function* be $F_P : \mathcal{D}_P \to \mathcal{D}_P$ satisfying the condition $S_P^D \circ \gamma \subseteq \gamma \circ F_P$. F_P is monotonic with respect to $\sqsubseteq$ and S_P^D is the immediate consequences operator mentioned in Section 2. Let the increasing sequence $Y_0, Y_1, \ldots$ be defined as follows. $Y_0 = \bot$, $Y_{n+1} = F_P(Y_n)$. These conditions are sufficient to establish that the limit of the sequence, say Y, satisfies $\gamma(Y) \supseteq \mathsf{lfp}(S_P^D) = M[\![P]\!]$ [12].

Since $\mathcal{D}_P$ contains infinite increasing chains, the sequence can be infinite. The use of a *widening* operator for convex polyhedra is needed to ensure convergence of the abstract interpretation. Define the sequence $Z_0 = Y_0$, $Z_{n+1} = Z_n \nabla F_P(Z_n)$ where ∇ is a widening operator for convex polyhedra [13]. The conditions on ∇ ensure that the sequence stabilises; thus for some finite j, $Z_i = Z_j$ for all $i > j$ and furthermore Z_j is an upper bound for the sequence $\{Y_i\}$. The value Z_j thus represents, via the concretisation function γ, an over-approximation of the least model of P. Furthermore much research has been done on improving the precision of widening operators, for example, widening-upto, or widening with thresholds [35, 36]. The widening upto operator ($\triangledown_T$) for convex polyhedra with respect to a set T of constraints (the threshold) is a widening operator $Z_1 \triangledown_T Z_2$ such that for all $\phi \in T$, $Z_1 \to \phi \wedge Z_2 \to \phi$ implies that $Z_1 \triangledown_T Z_2 \to \phi$. In other words the widening-upto operator preserves as many of the constraints in the threshold as possible.

3.1.1. Algorithm for convex polyhedral approximation of CHCs

Algorithm 1: Naive Algorithm for Convex Polyhedral Analysis

```
   Input: A set of CHCs $P$
   Output: over-approximation of the minimal model of $P$
 1 $i \leftarrow 0$ ;
 2 $Z_0 \leftarrow \bot$ ;
 3 $New \leftarrow \bot$ ;
 4 $Changed \leftarrow \{p \mid p \text{ is a predicate in } P\}$ ;
 5 while $Changed \neq \emptyset$ do
 6     foreach $(p(X) \leftarrow Body) \in P$ do
 7         $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
 8     $Z_{i+1} \leftarrow Z_i \nabla (New \sqcup Z_i)$ ;              /* Upper bound and widen */
 9     $Changed \leftarrow \{p \mid p \text{ has changed in } Z_{i+1}\}$ ;
10     $i \leftarrow i + 1$
11 return $Z_i$
```

Given the elements of convex polyhedral analysis summarised above, we present the algorithm for computing a polyhedral approximation of a set of CHCs. A naive algorithm to compute the limit of the sequence $Z_0, Z_1, Z_2, \ldots$ is given in Algorithm 1. This naive algorithm is just a stepping stone to present the main algorithm in Figure 2. Given a clause $p(X) \leftarrow Body$, the function call $\mathsf{solve}(p(X), Body, Z_i)$ returns a constrained fact $p(X) \leftarrow \phi$, where ϕ is the result of solving $Body$ in the current approximation Z_i (note that ϕ is a constraint in the theory of linear arithmetic). More precisely, if $Body = \psi, p_1(X_1), \ldots, p_r(X_r)$ then $\phi = (\psi \wedge \phi_1 \wedge \ldots \wedge \phi_r)|_X$, where $p_i(X_i) \leftarrow \phi_i$ (for $i = 1 \ldots k$) is a (renamed) constrained fact in Z_i. We assume that the constraint theory admits a projection operator, and we write $\phi|_X$ to mean the projection of ϕ onto the variables X. That is, if Y is the set of variables in ϕ and $Z = Y \setminus X$ then the variables in Z do not occur in $\phi|_X$ and

$\phi|_X \equiv \exists Z.\phi$.

Our algorithm, shown in Algorithm 2, incorporates generic optimisations for computing fixed points using an ascending chain. We present it in some detail since we are not aware of implementations that incorporate the same range of optimisations and precision enhancements, although all are drawn from the literature. The first step is to compute the strongly connected components (SCCs) of the predicate dependency graph of the set of CHCs. Each component is a set of (non-constraint) predicates; a group is either non-recursive (in which case it is a singleton) or a set of mutually recursive predicates. The algorithm for computing SCCs returns the components in topologically sorted order $C_1, \ldots, C_m$, such that for each C_j, no predicate in C_j depends on any predicate in C_k where $k > j$ [71].

Algorithm 2: Algorithm for Convex Polyhedral Analysis

```
Input: A set of CHCs $P$
Output: over-approximation of the minimal model of P
$C_1, \ldots, C_m \leftarrow$ SCCs for $P$ ;
$T \leftarrow \mathsf{thresholds}(P)$ ;
$i \leftarrow 0$ ;
$Z_0 \leftarrow \bot$ ;
for $j = 1$ to $m$ do
    if $C_j$ is recursive then
        $Changed \leftarrow \bigcup_{l=1..j} C_l$
        while $Changed \neq \emptyset$ do
            $New \leftarrow \bot$ ;
            foreach $(p(X) \leftarrow Body) \in P$ where $p \in C_j$ do
                if Body has changed in Changed then
                    $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
            $Z_{i+1} \leftarrow Z_i \nabla_T (New \sqcup Z_i)$ ;          /* Upper bound and widen */
            $Changed \leftarrow \{p \mid p$ has changed in $Z_{i+1}\}$ ;
            $i \leftarrow i + 1$
    else
        $New \leftarrow \bot$ ;
        foreach $(p(X) \leftarrow Body) \in P$ where $p \in C_j$ do
            $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
        $Z_{i+1} \leftarrow Z_i \sqcup New$ ;          /* Upper bound (no widening) */
        $i \leftarrow i + 1$
return $Z_i$
```

The algorithm proceeds to solve the components in order. A fixed point is computed for each SCC separately. A standard optimisation for recursive SCCs (the semi-naive optimisation) [73] is to keep track of which predicates have a new solution in each iteration. The set *Changed* records the predicates whose solution is changed. This optimisation allows a clause to be ignored on an iteration, if no predicate in its body has changed since the previous iteration. Obviously such a clause can contribute nothing new to the approximation. A recursive SCC is solved when the set *Changed* is empty after some iteration. For non-recursive SCCs, no iteration is needed.

The bodies of the clauses for the predicate in that SCC are solved with respect to the current approximation and their solutions are added to the current approximation.

We apply a widening-upto operator ∇_T where T contains a set of threshold constraints computed at the start of the algorithm (Algorithm 2, line 2), which we define in the next paragraph. T consists of facts that represent "guesses" for invariants for each predicate p. Any set T does not alter the soundness result (always produces an over-approximation of the minimal modle of P) of the Algorithm 2, but a good choice of thresholds can make a significant difference to the precision of the final result. In our implementation we adapt a method presented by Lakhdar-Chaouch *et al.* [53]. In brief, the method collects constraints by iterating the abstract semantic function F_P three times starting from the "top" ($\top$) element of $\mathcal{D}$, that is, the interpretation which assigns the universal polyhedron (the polyhedron representing the whole space of a given dimension or true constraint) to each predicate. The choice of three iterations is motivated by Lakhdar-Chaouch *et al.*; however, we believe that further experimentation with choices of thresholds would be fruitful.

We define the operation $\mathsf{thresholds}(P)$ as follows. First define a function which splits a constrained fact into a set of constrained facts having a single constraint. $\mathsf{atomconstraints}(p(Z) \leftarrow \phi)$ returns the set of constrained facts $\{p(Z) \leftarrow \phi_i \mid \phi = \phi_1 \wedge \ldots \wedge \phi_k,\ i = 1 \ldots k\}$ where ϕ_i are atomic constraints. The function is extended to apply to sets of constrained facts.

$$\mathsf{atomconstraints}(I) = \bigcup_{p(Z) \leftarrow \phi \in I} \{\mathsf{atomconstraints}(p(Z) \leftarrow \phi)\}.$$

Then define the thresholds function as follows.

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

Following this definition, the threshold constraints generated for our example program in Figure 2 is shown in Example 1.

Example 1 (Threshold Constraints)**.**

```
l(A,B) :- A=1.  l(A,B) :- B=0.  l(A,B) :- B=1.
l(A,B) :- A=2.  l(A,B) :- B=2.
false :- true.
```

3.2. *The query-answer transformation*

In Section 1.1 we discussed the origins and motivation of the query-answer transformation. In the following, we define it formally. We assume that, for each atom $A = p(t)$, A^{a} and A^{q} represent the atoms $p^{\mathsf{a}}(t)$ and $p^{\mathsf{q}}(t)$ respectively.

Definition 1 (Query-answer program)**.** *Given a set of CHCs P and an atom A, the (left-) query-answer clauses for P with respect to A, denoted* P^{qa}_A *or just* P^{qa}*, are as follows.*

- *(Answer clauses). For each clause* $H \leftarrow \phi, B_1, \ldots, B_n$ $(n \geq 0)$ *in P,* P^{qa} *contains the clause* $H^{\mathsf{a}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \ldots, B_n^{\mathsf{a}}$.
- *(Query clauses). For each clause* $H \leftarrow \phi, B_1, \ldots, B_i, \ldots, B_n$ $(n \geq 0)$ *in P,* P^{qa} *contains the following clauses:*

$$\begin{array}{l}
B_1^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}.\\
\dots\\
B_i^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \dots, B_{i-1}^{\mathsf{a}}.\\
\dots\\
B_n^{\mathsf{q}} \leftarrow \phi, H^{\mathsf{q}}, B_1^{\mathsf{a}}, \dots, B_{n-1}^{\mathsf{a}}.
\end{array}$$

- *(Goal clause).* $A^{\mathsf{q}} \leftarrow true.$

The clauses in P^{qa} encode a left-to-right, depth-first computation of the query $\leftarrow A$ for CHC clauses P (that is, the standard CLP computation rule, SLD extended with constraints). This is a complete proof procedure (produces a proof for each provable statement), assuming that all clauses matching a given call are explored in parallel. (Note: the incompleteness of standard CLP proof procedures arises due to the fact that clauses are tried in a fixed order). We can also define a query-answer transformation which encodes atoms in a right-to-left fashion. Since P^{qa} above encodes atoms in a left-to-right fashion, we call such a tranformation (left-) query answer transformation for clarity.

The answer clauses arise since there is an answer for the head predicate H if it was queried and all the body atoms have answers and ϕ holds. The query clauses arise since given a clause $H \leftarrow \phi, B_1, \dots, B_n$ $(n \geq 0)$, the i^{th} body atom B_i can only be queried if the head H is queried, ϕ holds and all the body atoms up to $i-1$ have answers (in the left-right computation). Finally, the goal clause asserts that the goal A is queried.

The size of query-answer program is quadratic with respect to the size of the original program. This is because we generate n query-answer clauses for each clause in the original program with n non-constraint atoms. So if we have m clauses in the original program and the maximum number of non-constraint atom in any clause is n, then the query-answer program contains at most $n * m + 1$ clauses.

Example 2 (Query-answer transformation)**.** *For a given predicate p, we represent p^a and p^q by `p_a` and `p_q` respectively in textual form. Given the program in Figure 2, its query-answer transformation following the Definition 1 is shown below. Note that the identifier preceding each clause shows the identifier of the original clause from where it is derived.*

```
%answer clauses
c1. l_a(A,B) :- l_q(A,B), A=1, B=0.
c2. l_a(A,B) :- l_q(A,B), A=C+D, B=D+1, l_a(C,D).
c3. false_a :- false_q, B>A, l_a(A,B).
%query clauses
c2. l_q(A,B) :- l_q(C,D), C=A+B, D=B+1.
c3. l_q(A,B) :- false_q, B>A.
%goal clause
false_q :-  true.
```

Query-answer clauses capture the mutual dependencies of top-down and bottom-up evaluation, since the queries and answers are defined in a single set of clauses. For example, given a clause $p \leftarrow q, p$, the call to p in the body depends on the answers for q (in a top-down left-right evaluation). However the answers for q depend on the calls to p in the head, since q is called from p. Top-down or bottom-up evaluation in isolation would not capture such mutual dependencies between calls and answers.

The relationship of the model of the clauses P^{qa} to the computation of the goal $\leftarrow A$ in P is expressed by the following property[1]. An SLD-derivation in CLP is a sequence $G_0, G_1, \ldots, G_k$ where each G_i is a goal $\leftarrow \phi, B_1, \ldots, B_m$, where ϕ is a constraint and $B_1, \ldots, B_m$ are atoms. In a left-to-right computation, G_{i+1} is obtained by resolving B_1 with a program clause. The model of P^{qa} captures (approximates) the set of atoms that are "called" or "queried" during the derivation, together with the answers (if any) for those calls. This is expressed precisely by Property 1.

Property 1 (Correctness of query-answer transformation)**.** *Let P be a set of CHCs and A be an atom. Let P^{qa} be the query-answer program for P wrt. A. Then*

(i) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \ \leftarrow A$ and $G_i = \ \leftarrow \phi, B_1, \ldots, B_m$, then $P^{\mathsf{qa}} \models \forall(B_1^{\mathsf{q}} \leftarrow \phi|_{\mathsf{vars}(B_1)})$;

(ii) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \ \leftarrow A$, containing a sub-derivation $G_{j_1}, \ldots, G_{j_k}$, where $G_{j_i} \leftarrow \phi', B, \mathcal{B}'$ and $G_{j_k} = \ \leftarrow \phi, \mathcal{B}'$, then $P^{\mathsf{qa}} \models \forall(B^{\mathsf{a}} \leftarrow \phi|_{\mathsf{vars}(B)})$. (This means that the atom B in G_{j_i} was successfully answered, with answer constraint $\phi|_{\mathsf{vars}(B)}$, where $\mathcal{B}'$ is a conjunction of atoms).

(iii) As a special case of (ii), if there is a successful derivation of the goal $\leftarrow A$ with answer constraint ϕ then $P^{\mathsf{qa}} \models \forall(A^{\mathsf{a}} \leftarrow \phi)$.

The correctness of query-answer transformation has already been established by several authors in the logic programming literature, for example, Nilsson [63] and Debray et al. [20].

4. Specialisation by constraint strengthening

We next present a procedure for specialising CHCs. In contrast to classical specialisation techniques based on partial evaluation with respect to a goal, the specialisation does not unfold the clauses at all; rather, it computes a specialised version of each clause, in which the constraints from the goal are propagated top-down and answers are propagated bottom-up.

We first make precise what is meant by "specialisation" for CHCs. Let P be a set of CHCs and let A be an atomic formula. The specialisation of P with respect to A is a set of clauses P_A such that for every constraint ϕ over the variables of A, $P \models \forall(A \leftarrow \phi)$ if and only if $P_A \models \forall(A \leftarrow \phi)$. This is a very general definition that allows for many transformations. In practice we are interested in specialisations that eliminate logical consequences of P that have no relevance to A.

For each clause $H \leftarrow \mathcal{B}$ in P, P_A contains a new clause $H \leftarrow \phi, \mathcal{B}$ where ϕ is a constraint. If the addition of ϕ makes the clause body unsatisfiable, it is the same as removing the clause, though removal is not essential to the procedure. Clearly P_A may have fewer logical consequences than P but our procedure guarantees that it preserves the logical consequences of P with respect to the (ground instances of) A. The procedure is summarised as follows: the inputs are a set of CHCs P and an atomic formula A.

1. Compute a *query-answer transformation* of P with respect to A, denoted P^{qa}, containing predicates p^{q} and p^{a} for each predicate p in P.

[1] Note that the model of P^{qa} might not correspond exactly to the calls and answers in the SLD-computation, since the CLP computation treats constraints as syntactic entities through decision procedures and the actual constraints could differ.

2. Compute an over-approximation M of the model of P^{qa}.
3. Strengthen the constraints in the clauses in P by adding constraints from the answer predicates in M.

Next we will explain each step in detail.

4.1. The query-answer transformation

This was presented in Section 3.2. We perform a query-answer transformation of P with respect to the goal *false*. We call the result P^{qa}. It follows from Property 1(iii) that if *false* is derivable from P then $false^{\mathsf{a}}$ is derivable from P^{qa}.

4.2. Over-approximation of the model of P^{qa}

Abstract interpretation of P^{qa} yields an over-approximation of $M[\![P^{\mathsf{qa}}]\!]$, say M, containing constrained facts for the query and answer predicates. These represent the calls and answers generated during all derivations starting from the goal A. In our experiments we use a convex polyhedral approximation (CPA) of $M[\![P^{\mathsf{qa}}]\!]$, as described in Section 3.1. Using CPA, we derive the following constrained facts for the program in Example 2.

Example 3 (Over-approximation of the model of the program in Example 2)**.**

```
false_q :- true.
l_(A,B) :- true.
l_a(A,B) :- A>=1, A-B>=0, B>=0.
```

For all predicates in a program for which the model contains no constrained fact, we assume that there is a constrained fact for that predicate whose right hand side contains an unsatisfiable constraint.

4.3. Strengthening the constraints in P

We use the information in the model of P^{qa}, say M, to specialise the original clauses in P. Suppose M contains constrained facts $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$ and $p^{\mathsf{a}}(X) \leftarrow \phi^{\mathsf{a}}$. (If there is no constrained fact $p^*(X) \leftarrow \phi$ for some p^* then we consider M to contain $p^*(X) \leftarrow false$, as mentioned above).

Given such a set M, define γ_M to be the mapping from atoms to constraints such that $\gamma_M(p^*(X)) = \phi$ for each constrained fact $p^*(X) \leftarrow \phi$, where $*$ is a or q.

Definition 2 (Strengthened clauses P_A from a model.)**.** *Let P be a set of CHCs, A be a goal and P^{qa} be the query-answer transformation of P with respect to A. Let M be a model of P^{qa} defined by a set of constrained facts. Then P_A contains the following clauses:*

$$P_A = \{p(X) \leftarrow \phi, \phi_0, \phi_1, \ldots, \phi_n, p_1(X_1), \ldots, p_k(X_k) \mid \begin{array}{l} p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k) \in P, \\ \phi_0 = \gamma_M(p^{\mathsf{a}}(X)), \phi_i = \gamma_M(p_i^{\mathsf{a}}(X_i)), \\ \mathsf{SAT}(\phi \wedge \phi_0 \wedge \bigwedge_{i=1}^{n} \phi_i) \} \end{array}$$

The clauses whose body constraints are unsatisfiable are removed from P_A, since they cannot contribute to feasible derivations (a Horn clause derivation whose constraints are unsatisfiable) and do not contribute to the minimal model of P_A. Here we assume that there is exactly one constrained fact in M for each predicate $p^a, p_1^a, \ldots, p_n^a$. Due to the choice of domain for abstract interpretation, we get one constrained fact (a convex polyhedron) for each predicate in the program. Using a richer domain such as the power set of convex polyhedra, we could obtain disjunctive constraints, which could be eliminated from the specialised clauses by program transformation. For example, the clause $\mathtt{p(X)} \leftarrow (\mathtt{X} > \mathtt{Y} \vee \mathtt{Y} < \mathtt{X}), \mathtt{q(Y)}$ can be transformed into $\mathtt{p(X)} \leftarrow \mathtt{X} > \mathtt{Y}, \mathtt{q(Y)}$ and $\mathtt{p(X)} \leftarrow \mathtt{Y} < \mathtt{X}, \mathtt{q(Y)}$. However, that could cause blow-up in the number of clauses generated.

Note that wherever M contains constrained facts $p^{\mathsf{a}}(X) \leftarrow \phi^{\mathsf{a}}$ and $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$, we have $\phi^{\mathsf{a}} \rightarrow \phi^{\mathsf{q}}$ since the answers for p are always stronger than the calls to p. Thus it suffices to add only the answer constraints to the clauses in P and we can ignore the model of the query predicates. A special case of this is where M contains a constrained fact $p^{\mathsf{q}}(X) \leftarrow \phi^{\mathsf{q}}$ but there is no constrained fact for $p^{\mathsf{a}}(X)$, or in other words M contains the constrained fact $p^{\mathsf{a}}(X) \leftarrow$ *false* (meaning that no ground atom exists for the predicate p^{a} in M). This means that all derivations for $p(X)$ fail or loop in P and so adding the answer constraint *false* for p eliminates looping derivations for p.

Example 4 (Constraint specialisation)**.** *Using the model of the query-answer transformed program presented in Example 3, the program in Figure 2 can be strengthened as follows:*

$\texttt{c1. l(A,B) :- A=1, B=0, }\underline{\texttt{A>=1, A-B>=0, B>=0}}\texttt{.}$

$\texttt{c2. l(A,B) :- A=C+D, B=D+1, }\underline{\texttt{A>=1, A-B>=0, B>=0}}\texttt{, }\underline{\texttt{C>=1, C-D>=0, D>=0}}\texttt{, l(C,D).}$

$\texttt{c3. false :- B>A, }\underline{\texttt{A>=1, A-B>=0, B>=0}}\texttt{, l(A,B).}$

The constraints in the clauses are strengthened by the addition of extra constraints from the model, which are underlined. It can be seen that the constraints in the body of integrity constraint (`c3`) are unsatisfiable, and thus `c3` can be eliminated.

Specialisation by strengthening the constraints preserves the answers of the goal with respect to which the query-answer transformation was performed. In particular, we have the following property.

Property 2 (Soundness of constraint specialisation)**.** *If P is a set of CHCs and P_A is the set of clauses obtained by strengthening the clause constraints as just described, then $P \models (A \leftarrow \phi)$ if and only if $P_A \models (A \leftarrow \phi)$.*

Proof. The proof follows from the standard Theorems (Theorem 6.0.1, Part 4 and Theorem 6.0.1, Part 2 of [39]) and Lemmas (Lemma 3.1 of [59]) from the literature in constraint logic programming.

□

The specialisation and analysis are separate in our approach. More complex algorithms intertwining them can be envisaged, though the benefits are not clear. Iteration of our procedure yields further specialisation, as our experiments confirm.

5. Application to the CHC verification problem

In this section we will briefly discuss the origin of Horn clauses in verification problems and then discuss the application of our constraint specialisation in Horn clause verification.

5.1. Origin of Horn clauses in program verification problems

CHCs provide a logical formalism suitable for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). Program verification usually refers to verification of source programs rather than some intermediate semantic form. Instead of devising verification procedure for each source language (which is a difficult process) we devise a verification procedure for CHCs and then translate source languages to it, saving time and effort. The literature on program analysis and verification contains several methods for generating CHCs from an imperative program, which fall into two broad categories.

1. *Specialising an interpreter*: The translation is usually obtained through specialisation of an interpreter (equivalently partial evaluation). Let P_{imp} be an imperative program written in language L and I be an interpreter of L written in some language (in Horn logic as Horn clauses). Partial evaluation or specialisation of I with respect to P_{imp} produces a specialised interpreter I_s for P_{imp}. I_s can be regarded as the translation of P_{imp} to the language in which the interpreter is written which preserves the semantics of P_{imp}. This approach is taken by Peralta *et al.* [64] and De Angelis, Fioravanti, Pettorossi and Proietti [19].
2. *Hoare style proof rules*: The translation is obtained by applying the proof rules to obtain logical proof subgoals whose satisfiability implies correctness of the original program. This approach is taken by Gurfinkel *et al.* [33], Grebenshchikov, Lopes, Popeea and Rybalchenko [30] and McMillan and Rybalchenko [61].

In both of these, the program semantics can be *small-step*, *big-step* or *mixed*. In the first category this is specified by an interpreter whereas in the second case it is specified by proof rules. The outcome of the translation in both cases is a set of Horn clauses often called *verification conditions* in the literature [19, 61]. There are also other ad-hoc techniques for translation, for example, [21, 41].

We illustrate this via the example program from Figure 1, which is taken from [16]. Suppose we would like to prove the Hoare triple $\{a = 1, b = 0\}$ `Loop_add` $\{a \geq b\}$. This means starting from a state satisfying $\{a = 1, b = 0\}$, if we execute `Loop_add` and if it terminates then the resulting state satisfies $\{a \geq b\}$. Let `l(a,b)` be an unknown loop invariant for the *while loop* in the program `Loop_add`, the above triple holds if the verification conditions in Figure 1b are satisfiable, that is, there exists an interpretation that satisfies each clause.

These conditions are pure logical formulas and are indeed a set of recursive Horn clauses. Thus, the problem of proving whether the program is safe (equivalently has a model) is reduced to checking whether this set of Horn clauses is satisfiable. Using CLP syntax the verification conditions can be written as shown in Figure 2. Each clause in this example is assigned an identifier (for example `c1` to `c3`) in order to refer them later. The clause `c3` is called an integrity constraint.

5.2. CHC verification

In this section, we discuss the application of our constraint specialisation in Horn clause verification. We assume that there is a distinguished predicate symbol *false* in P which is always 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. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$ is equivalent to the formula $false \leftarrow \neg\phi_1 \wedge \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$. The latter might not be a CHC (e.g. if ϕ_1 contains =) but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula $X = Y \leftarrow p(X, Y)$ is equivalent to the set of CHCs $\{false \leftarrow X > Y, p(X, Y),\ false \leftarrow X < Y, p(X, Y)\}$.

Integrity constraints can be seen as safety properties (nothing bad will happen, which we will define formally in the next section). For example if a set of CHCs encodes the behaviour of a system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses. In Figure 2 the verification problem focuses on proving that the integrity constraint is satisfied. This can only happen if the body of `c3` is unsatisfiable. A program is considered safe if its verification conditions have a model.

5.3. The CHC verification problem.

To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. If so we say that P is safe. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the logical consequence relation. 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 $P \not\models$ 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. The verification problem can be formulated deductively rather than model-theoretically. We can exploit proof procedures for constraint logic programming [38] to reason about the satisfiability of a set of CHCs.

5.3.1. Proof Techniques for Horn clauses

The techniques that we use in this paper are:

- *Proof by over-approximation of the minimal model:* Given a set of CHCs, its minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P [58]. That is, $P \models p(a)$ if and only if $p(a) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathit{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathit{false} \notin M'$. This technique is called proof by *over-approximation of the minimal model*.

- *Proof by specialisation:* In our context we use specialisation to focus the verification problem on the formula to be proved. More specifically, we specialise a set of CHCs with respect to a "query" to the atom *false*; thus the specialised CHCs entail *false* if and only if the original clauses entailed *false*. The constraint strengthening procedure described in Section 4 is our method of specialisation. So whenever we refer to specialisation we refer to this method unless otherwise stated.

5.3.2. Analysis of the specialised clauses

Having specialised the clauses with respect to *false*, it may be that the clauses P_{false} do not contain a clause with head *false*. In this case P_{false} is safe, since clearly this is a sufficient condition for $P_{\mathsf{false}} \not\models \mathit{false}$. This is the case for our example program since the body of the clause `c3` is unsatisfiable after constraint strengthening, it is removed from the set of specialised clauses.

If this check fails we still do not know whether P has a model. In this case we can perform the convex polyhedral analysis on the clauses P_{false}. As the experiments later show, safety is often provable by checking the resulting model; if no constrained fact for *false* is present, then $P_{\mathsf{false}} \not\models false$. If safety is not proven, there are two possibilities: the approximate model is not precise enough, but P has a model, or there is a proof of *false*. Refinement techniques could be used to distinguish these, but this is not the topic of this paper.

In summary, our experimental procedure for evaluating the effectiveness of constraint specialisation contains two steps. Given a set of CHCs P with integrity constraints: (1) Compute a specialisation of P with respect to *false* yielding P_{false}. If P_{false} contains no integrity constraints, then P is safe. (2) If P_{false} does contain integrity constraints, perform a convex polyhedra analysis of P_{false}. If the resulting approximation of the minimal model contains no constrained fact for the predicate *false*, then P_{false} is safe and hence P is safe. If we find a concrete feasible derivation for *false* then we conclude that P is unsafe. Otherwise, P is possibly unsafe. Please refer to [46] for deriving traces using convex polyhedral approximation.

6. Experimental evaluation

Table 1 presents experimental results of applying our constraint specialisation to a number of Horn clause verification benchmarks taken from the repository of Horn clauses [5] and other sources including [28, 40, 32, 6, 17]. The columns CPA, QARMC [29] and Eldarica [37] present the results of verification using convex polyhedra, QARMC and Eldarica respectively, whereas columns CS + CPA, CS + QARMC and CS + Eldarica show the result of running constraint specialisation followed by CPA, QARMC and Eldarica respectively. The symbol "-" is used in the table to indicate that the result is not significant in the given case. The experiments were carried out on an Intel(R) X5355 quad-core (@ 2.66GHz) computer with 6 GB memory running Debian 5. We set 5 minutes of timeout for each experiment. The specialisation procedure is implemented in the tool called Rahft which is publicly available from `https://github.com/bishoksan/RAHFT/`. The tool offers a simple command line interface and accepts options for constraint specialisation. For this purpose it can be run using the command: `./rahft input -sp output` where the `input` is a set of CHCs and `output` is a file name to store the specialised CHCs and `-sp` is an option for clause specialisation. The benchmark programs are available from `https://github.com/bishoksan/RAHFT/tree/master/benchmarks_scp`.

The results show that constraint specialisation is effective in practice. We report that 109 out of 218, that is 50%, of the problems are solved by constraint specialisation alone. When used as a pre-processor for other verification tools, the results show improvements on both the number of instances solved and the solution time. Using our tool, we report approximately 47% increase in the number of instances solved and twice as fast on average. Using QARMC, we report 13% increase in the number of instances solved and 5 times faster on average. Similarly using Eldarica, we report approximately 12% increase in the number of instances solved and almost 4 times faster on average. It is important to note that there is no refinement iteration in CPA as there is in QARMC and Eldarica.

	CPA	CS + CPA	QARMC	CS + QARMC	Eldarica	CS + Eldarica
solved (safe/unsafe)	61 (48/13)	162 (144/18)	178 (141/37)	205 (171/34)	159(135/24)	206 (175/31)
unknown / timeout	144/13	49/7	-/40	-/13	-/59	-/12
total time (secs)	2317	1303	13367	2613	10805	3235
average time (secs)	10.62	5.97	61.31	11.98	50.02	14.97
%solved	27.98	74.31	81.65	94.04	73	95.3

Table 1: Experiments on a set of 218 (181 safe and 37 unsafe) CHC verification problems with a timeout of five minutes

The (perhaps surprising) effectiveness of this relatively simple combination of constraint specialisation and convex polyhedral analysis is underlined by noting that it can solve problems for which more complex methods have been proposed. For example, apart from the many examples from the Horn clause verification benchmarks that require refinement using CEGAR-based approaches, the technique solves the "rate-limiter" and "Boustrophedon" examples presented by Monniaux and Gonnord [62] (Section 5) (directly encoded as Horn clauses); their approach, also based on convex polyhedra, uses bounded model checking to achieve a partitioning of the approximation, while other approaches to such problems use trace-partitioning and look-ahead widening.

It is possible to strengthen constraints in the clauses using the model of the original program (denote it by CPA') rather than its query-answer transformed one. The effect of such a specialisation (CS + CPA') on these set of benchmarks is same as applying CPA directly on the original programs, but such a specialisation may be a useful pre-processing for other tools. For example, the following program (a variant of our example program) is not solved by such a combination (CS + CPA') but our current approach does (CS + CPA).

```
false :- A=1, B=0, l(A,B).
l(A,B) :- C=A+B, D=B+1, l(C,D).
l(A,B) :- B>A.
```

We were able to solve almost 100 more problems with our proposed approach. Therefore the role of query-answer transformation is crucial for propagating constraints in verification problems. As mentioned earlier, our specialisation procedure can be iterated which yields further specialisation of the clauses. By iterating the procedure, we were able to solve 12 more problems. After second iteration, we observed that the same program was produced in most of the cases indicating that the successive iterations do not produce any further specialisation.

6.1. Additional experiments on SV-COMP-15 benchmarks

We chose a subset of 132 problems, written in C, from SV-COMP 2015[2] [7]. 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* and *loop-new*. We used SeaHorn [34, 33], a verification framework based on LLVM, for generating Horn clause from C programs. SeaHorn first compiles C to LLVM intermediate representation (LLVM IR), also known as bitcode using *clang*, a C-family front-end for LLVM[3]. 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. Gurfinkel et al. [33] have shown that some of the problems are solved by these transformations only. 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 [34, 33]. The benchmark programs are available from `https://github.com/bishoksan/RAHFT/tree/master/benchmarks_scp`. The results are summarised in the Table 2. They again show that our method, the constraint specialisation, is in fact effective in practice since we can solve 18% more problems using it as a pre-processor to our convex polyhedra tool.

[2]`http://sv-comp.sosy-lab.org/2015/benchmarks.php`

[3]`http://clang.llvm.org/`

	CPA	CS + CPA
solved (safe/unsafe)	37 (14/23)	61 (26/35)
unknown	95	71
average time (secs.)	0.51	0.50
solved (%)	28	46

Table 2: Experimental results on 132 CHC verification problems with a timeout of five minutes

7. Conclusion and Future Work

We introduced a method for specialising the constraints in constrained Horn clauses with respect to a goal. The method is based a combination of techniques that have already shown their usefulness, especially abstract interpretation and query-answer transformation. The particular combination of techniques we chose was arrived at by experimentation and analysis of the needs of the problem. The approach propagates constraints globally, both forwards and backwards, and produces explicit invariants from the original clauses.

We applied the method to program verification problems encoded as constrained Horn clauses. Experiments showed firstly that constraint specialisation alone is an effective verification tool. Secondly, it can be applied as a pre-processor, improving the effectiveness of other verification tools. It remains to be checked if the solver like VeriMap would benefit from our specialisation.

The effectiveness of the procedure is at first sight somewhat surprising. Its effectiveness comes from the fact that it focuses on full exploitation of the available information, propagating information simultaneously top-down and bottom-up, and the use of powerful analysis techniques based on abstract interpretation capable of discovering useful invariants. Moreover the addition of the widening-upto method with threshold generation plays an important role in the procedure. Care was taken to implement the procedures efficiently.

The technique is independent of the constraint theory underlying the clauses and the abstract domain for analysis, although we only experimented so far with the domain of linear arithmetic constraints, and the domain of convex polyhedra.

Future work. There is potential for applying this technique in future work whenever explicit constraints need to be extracted from clauses. One such instance is in program debugging since more specific information may make errors in the original program apparent. Another is as a pre-processor in program specialisation where knowledge of the call context of each program point could enable specialisations which are not otherwise obviously available. Finally, termination and resource analysis could benefit from constraint specialisation, since these might enable better ranking functions to be discovered, proving decrease of some expression in each loop.

The query-answer transformation has several variations, which can give differing precision when combined with abstract interpretation. For instance, more refined query predicates of the form $p^{\mathsf{q}}_{i,j}$ could be generated representing calls to the i^{th} atom in the body of clause j [26]. Secondly, the left-to-right computation could be replaced by right-to-left or any other order. The success or failure of a goal is independent of the computation rule; hence we could generate answers using other computation rules, or combining computation rules [27]. While different computation rules do not affect the model of the answer predicates, more effective propagation of constraints during program analysis, and thus greater precision, can sometimes be achieved by varying the computation rule.

Acknowledgements

This paper is an extended version of a paper presented at the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'15). We thank the anonymous referees of PEPM'15 for useful comments as well as the referees for this journal, who gave many constructive suggestions for improvement. The paper has been extended in several directions. We have provided algorithms and implementation details, extended the related work and other sections. We have also carried out some further experiments in a set of examples from software verification benchmarks available in the literature and evaluated our constraint specialisation as a pre-processor to other Horn clause verification tools in the literature. We would like to thank Jorge A. Navas for his help with the tool SeaHorn and for several interesting discussions and Emanuele De Angelis for providing us with benchmark programs. 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] Baier, C., Tinelli, C. (Eds.), 2015. 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.
URL `http://dx.doi.org/10.1007/978-3-662-46681-0`

[2] Ball, T., Levin, V., Rajamani, S. K., 2011. A decade of software model checking with SLAM. Commun. ACM 54 (7), 68–76.

[3] Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J., 1986. Magic sets and other strange ways to implement logic programs. In: Proceedings of the 5^{th} ACM SIGMOD-SIGACT Symposium on Principles of Database Systems.

[4] Benoy, F., King, A., August 1996. Inferring argument size relationships with CLP(R). In: Gallagher, J. P. (Ed.), Logic-Based Program Synthesis and Transformation (LOPSTR'96). Vol. 1207 of LNCS. pp. 204–223.

[5] Beyer, D., Nov. 2012. Repository of Horn clauses. `https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses`, accessed: 2016-04-18.

[6] Beyer, D., 2013. Second competition on software verification - (summary of sv-comp 2013). In: Piterman, N., Smolka, S. A. (Eds.), TACAS. Vol. 7795 of Lecture Notes in Computer Science. Springer, pp. 594–609.
URL `http://dx.doi.org/10.1007/978-3-642-36742-7`

[7] Beyer, D., 2015. Software verification and verifiable witnesses - (report on SV-COMP 2015). In: [1], pp. 401–416.
URL `http://dx.doi.org/10.1007/978-3-662-46681-0_31`

[8] Bjørner, N., McMillan, K. L., Rybalchenko, A., 2013. On solving universally quantified Horn clauses. In: Logozzo, F., Fähndrich, M. (Eds.), SAS. Vol. 7935 of LNCS. Springer, pp. 105–125.

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

[10] Codish, M., Demoen, B., 1995. Analyzing logic programs using "`PROP`"-ositional logic programs and a magic wand. J. Log. Program. 25 (3), 249–274.

[11] Cousot, P., Cousot, R., 1977. 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. ACM, pp. 238–252.

[12] Cousot, P., Cousot, R., 1992. Abstract interpretation frameworks. Journal of Logic and Computation 2 (4), 511–547.

[13] Cousot, P., Halbwachs, N., 1978. 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.

[14] Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., Sekar, R. C., 1993. Extracting determinacy in logic programs. In: Warren, D. S. (Ed.), Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, June 21-25, 1993. MIT Press, pp. 424–438.

[15] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M., 2013. Verifying programs via iterated specialization. In: Albert, E., Mu, S.-C. (Eds.), PEPM. ACM, pp. 43–52.

[16] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M., 2014. Program verification via iterated specialization. Sci. Comput. Program. 95, 149–175.
URL `http://dx.doi.org/10.1016/j.scico.2014.05.017`

[17] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M., 2014. Verimap: A tool for verifying programs through transformations. In: Ábrahám, E., Havelund, K. (Eds.), TACAS. Vol. 8413 of LNCS. Springer, pp. 568–574.

[18] De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M., 2015. Proving correctness of imperative programs by linearizing constrained horn clauses. TPLP 15 (4-5), 635–650.
URL `http://dx.doi.org/10.1017/S1471068415000289`

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

[20] Debray, S. K., Ramakrishnan, R., 1994. Abstract interpretation of logic programs using magic transformations. J. Log. Program. 18 (2), 149–176.
URL `http://dx.doi.org/10.1016/0743-1066(94)90050-7`

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

[22] Fujita, H., 1987. An algorithm for partial evaluation with constraints. Tech. Rep. TR-258, ICOT.

[23] Gallagher, J. P., June 1993. Specialisation of logic programs: A tutorial. In: Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation. ACM Press, Copenhagen, pp. 88–98.

[24] Gallagher, J. P., Bruynooghe, M., 1990. Some low-level source transformations for logic programs. In: Proceedings of Meta90 Workshop on Meta Programming in Logic. Katholieke Universiteit Leuven, Belgium.

[25] Gallagher, J. P., Codish, M., Shapiro, E., 1988. Specialisation of Prolog and FCP programs using abstract interpretation. New Generation Computing 6, 159–186.

[26] Gallagher, J. P., de Waal, D., 1993. Deletion of redundant unary type predicates from logic programs. In: Lau, K., Clement, T. (Eds.), Logic Program Synthesis and Transformation. Workshops in Computing. Springer-Verlag, pp. 151–167.

[27] Gallagher, J. P., de Waal, D., 1994. Fast and precise regular approximation of logic programs. In: Van Hentenryck, P. (Ed.), Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy. MIT Press.

[28] Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., Stuckey, P. J., 2013. Failure tabled constraint logic programming by interpolation. TPLP 13 (4-5), 593–607.

[29] Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., Rybalchenko, A., 2012. Hsf(c): A software verifier based on Horn clauses - (competition contribution). In: Flanagan, C., König, B. (Eds.), TACAS. Vol. 7214 of LNCS. Springer, pp. 549–551.

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

[31] Gupta, A., Popeea, C., Rybalchenko, A., 2011. Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (Ed.), Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings. Vol. 7078 of Lecture Notes in Computer Science. Springer, pp. 188–203.
URL `http://dx.doi.org/10.1007/978-3-642-25318-8_16`

[32] Gupta, A., Rybalchenko, A., 2009. Invgen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (Eds.), CAV. Vol. 5643 of LNCS. Springer, pp. 634–640.

[33] Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J. A., 2015. The seahorn verification framework. In: Kroening, D., Pasareanu, C. S. (Eds.), Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I. Vol. 9206 of Lecture Notes in Computer Science. Springer, pp. 343–361.
URL `http://dx.doi.org/10.1007/978-3-319-21690-4_20`

[34] Gurfinkel, A., Kahsai, T., Navas, J. A., 2015. Seahorn: A framework for verifying C programs (competition contribution). In: [1], pp. 447–450.
URL `http://dx.doi.org/10.1007/978-3-662-46681-0_41`

[35] Halbwachs, N., Proy, Y. E., Raymound, P., September 1994. Verification of linear hybrid systems by means of convex approximations. In: Proceedings of the First Symposium on Static Analysis. Vol. 864 of LNCS. pp. 223–237.

[36] Halbwachs, N., Proy, Y. E., Raymound, P., 1997. Verification of real-time systems using linear relation analysis. Formal Methods in System Design 11 (2), 157–185.

[37] Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P., 2012. A verification toolkit for numerical

transition systems - tool paper. In: Giannakopoulou, D., Méry, D. (Eds.), FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings. Vol. 7436 of Lecture Notes in Computer Science. Springer, pp. 247–251.
URL `http://dx.doi.org/10.1007/978-3-642-32759-9_21`

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

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

[40] Jaffar, J., Navas, J. A., Santosa, A. E., 2011. Unbounded symbolic execution for program verification. In: Khurshid, S., Sen, K. (Eds.), RV. Vol. 7186 of LNCS. Springer, pp. 396–411.

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

[42] Jones, N., Gomard, C., Sestoft, P., 1993. Partial Evaluation and Automatic Software Generation. Prentice Hall.

[43] Jones, N., Mycroft, A., 1986. Dataflow analysis of applicative programs using minimal function graphs. In: Proceedings of Principle of Programming Languages (POPL'86). ACM Press.

[44] Jones, N. D., 1997. Combining abstract interpretation and partial evaluation. In: Van Hentenryck, P. (Ed.), Symposium on Static Analysis (SAS'97). Vol. 1302 of Springer-Verlag Lecture Notes in Computer Science. pp. 396–405.

[45] Jones, N. D., Rosendahl, M., 1997. Higher-order minimal function graphs. Journal of Functional and Logic Programming 1997(2).

[46] Kafle, B., Gallagher, J. P., 2016. Interpolant tree automata and their application in horn clause verification. In: Hamilton, G. W., Lisitsa, A., Nemytykh, A. P. (Eds.), Proceedings of the Fourth International Workshop on Verification and Program Transformation, VPT@ETAPS 2016, Eindhoven, The Netherlands, 2nd April 2016. Vol. 216 of EPTCS. pp. 104–117.
URL `http://dx.doi.org/10.4204/EPTCS.216.6`

[47] Kafle, B., Gallagher, J. P., Ganty, P., 2016. Solving non-linear horn clauses using a linear horn clause solver. In: Gallagher, J. P., Rümmer, P. (Eds.), Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016. Vol. 219 of EPTCS. pp. 33–48.
URL `http://dx.doi.org/10.4204/EPTCS.219.4`

[48] Kafle, B., Gallagher, J. P., Morales, J. F., 2016. Rahft: A tool for verifying horn clauses using abstract interpretation and finite tree automata. In: Chaudhuri, S., Farzan, A. (Eds.), Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I. Vol. 9779 of Lecture Notes in Computer Science. Springer, pp. 261–268.
URL `http://dx.doi.org/10.1007/978-3-319-41528-4_14`

[49] Klimov, A. V., 2011. Solving coverability problem for monotonic counter systems by supercompilation. In: Clarke, E. M., Virbitskaite, I., Voronkov, A. (Eds.), Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers. Vol. 7162 of Lecture Notes in Computer Science. Springer, pp. 193–209.
URL `http://dx.doi.org/10.1007/978-3-642-29709-0_18`

[50] Klyuchnikov, I. G., Romanenko, S. A., 2009. Proving the equivalence of higher-order terms by means of supercompilation. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (Eds.), Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers. Vol. 5947 of Lecture Notes in Computer Science. Springer, pp. 193–205.
URL `http://dx.doi.org/10.1007/978-3-642-11486-1_17`

[51] Komorowski, H. J., 1992. An introduction to partial deduction. In: Pettorossi, A. (Ed.), META. Vol. 649 of LNCS. Springer, pp. 49–69.

[52] Lafave, L., Gallagher, J. P., 1998. Partial evaluation of functional logic programs in rewriting-based languages. In: Fuchs, N. (Ed.), Logic Program Synthesis and Transformation (LOPSTR'97). Springer-Verlag Lecture Notes in Computer Science.

[53] Lakhdar-Chaouch, L., Jeannet, B., Girault, A., 2011. Widening with thresholds for programs with complex control graphs. In: Bultan, T., Hsiung, P.-A. (Eds.), ATVA 2011. Vol. 6996 of LNCS. Springer, pp. 492–502.

[54] Leuschel, M., 1999. Advanced logic program specialisation. In: Hatcliff, J., Mogensen, T. Æ., Thiemann, P. (Eds.), Partial Evaluation - Practice and Theory. Vol. 1706 of LNCS. Springer, pp. 271–292.

[55] Leuschel, M., 2004. A framework for the integration of partial evaluation and abstract interpretation of logic programs. ACM Trans. Program. Lang. Syst. 26 (3), 413–463.
URL `http://doi.acm.org/10.1145/982158.982159`

[56] Leuschel, M., Massart, T., 1999. Infinite state model checking by abstract interpretation and program specialisation.

In: Bossi, A. (Ed.), LOPSTR'99. Vol. 1817 of LNCS. Springer, pp. 62–81.

[57] Lisitsa, A., Nemytykh, A. P., 2008. Reachability analysis in verification via supercompilation. Int. J. Found. Comput. Sci. 19 (4), 953–969.
URL http://dx.doi.org/10.1142/S0129054108006066

[58] Lloyd, J., 1987. Foundations of Logic Programming: 2nd Edition. Springer-Verlag.

[59] Marriott, K., Naish, L., Lassez, J.-L., 1988. Most specific logic programs. In: Proc. Fifth International Conference on Logic programming, Seattle, WA. MIT Press.

[60] Marriott, K., Stuckey, P. J., 1993. The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering. In: Deusen, M. S. V., Lang, B. (Eds.), Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993. ACM Press, pp. 334–344.
URL http://doi.acm.org/10.1145/158511.158685

[61] McMillan, K. L., Rybalchenko, A., 2013. Solving constrained horn clauses using interpolation. Tech. rep., Microsoft Research.

[62] Monniaux, D., Gonnord, L., 2011. Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (Ed.), SAS. Proceedings. Vol. 6887 of Lecture Notes in Computer Science. Springer, pp. 369–385.
URL http://dx.doi.org/10.1007/978-3-642-23702-7_27

[63] Nilsson, U., 1995. Abstract interpretation: A kind of magic. Theor. Comput. Sci. 142 (1), 125–139.
URL http://dx.doi.org/10.1016/0304-3975(94)00223-6

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

[65] Peralta, J. C., Gallagher, J. P., 2002. Convex hull abstractions in specialization of CLP programs. In: Leuschel, M. (Ed.), LOPSTR. Vol. 2664 of LNCS. Springer, pp. 90–108.

[66] Pettorossi, A., Proietti, M., 2000. Perfect model checking via unfold/fold transformations. In: Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L. M., Sagiv, Y., Stuckey, P. J. (Eds.), Computational Logic. Vol. 1861 of LNCS. Springer, pp. 613–628.

[67] Podelski, A., Rybalchenko, A., 2007. ARMC: the logical choice for software model checking with abstraction refinement. In: Hanus, M. (Ed.), PADL 2007. Vol. 4354 of LNCS. pp. 245–259.

[68] Ramakrishnan, R., 1988. Magic templates: A spellbinding approach to logic programs. In: Kowalski, R. A., Bowen, K. A. (Eds.), Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes). MIT Press, pp. 140–159.

[69] Rümmer, P., Hojjat, H., Kuncak, V., 2013. Disjunctive interpolants for Horn-clause verification. In: Sharygina, N., Veith, H. (Eds.), CAV. Vol. 8044 of Lecture Notes in Computer Science. Springer, pp. 347–363.
URL http://dx.doi.org/10.1007/978-3-642-39799-8

[70] Srivastava, D., Ramakrishnan, R., 1993. Pushing constraint selections. J. Log. Program. 16 (3), 361–414.
URL http://dx.doi.org/10.1016/0743-1066(93)90048-L

[71] Tarjan, R. E., 1972. Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2), 146–160.

[72] Turchin, V. F., 1980. The use of metasystem transition in theorem proving and program optimization. In: de Bakker, J. W., van Leeuwen, J. (Eds.), Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings. Vol. 85 of LNCS. Springer, pp. 645–657.
URL http://dx.doi.org/10.1007/3-540-10003-2_105

[73] Ullman, J., 1988. Principles of Knowledge and Database Systems; Volume 1. Computer Science Press.

[74] Vieille, L., 1986. Recursive axioms in deductive databases: The query/subquery approach. In: Expert Database Conf. pp. 253–267.

[75] Winsborough, W. H., 1992. Multiple specialization using minimal-function graph semantics. J. Log. Program. 13 (2&3), 259–290.