Predicate Pairing with Abstraction for Relational Verification*

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2,3], and Maurizio Proietti[3]

[1] DEC, University 'G. d'Annunzio', Chieti-Pescara, Italy
{emanuele.deangelis,fabio.fioravanti}@unich.it
[2] DICII, University of Rome 'Tor Vergata', Italy
pettorossi@info.uniroma2.it
[3] IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. Relational verification is a technique that aims at proving properties that relate two different program fragments, or two different program runs. It has been shown that constrained Horn clauses (CHCs) can effectively be used for relational verification by applying a CHC transformation, called *Predicate Pairing*, which allows the CHC solver to infer relations among arguments of different predicates. In this paper we study how the effects of the Predicate Pairing transformation can be enhanced by using various abstract domains based on Linear Arithmetic (i.e., the domain of convex polyhedra and some of its subdomains) during the transformation. After presenting an algorithm for Predicate Pairing with abstraction, we report on the experiments we have performed on over a hundred relational verification problems by using various abstract domains. The experiments have been performed by using the VeriMAP verification system, together with the Parma Polyhedra Library (PPL) and the Z3 solver for CHCs.

1 Introduction

Relational program properties are properties that relate two different programs or two executions of the same program. Relational properties that have been studied in the literature include program equivalence, non-interference for software security, and relative correctness [4, 5, 22].

Recent papers have advocated the use of *Constrained Horn Clauses* (CHCs) for the verification of relational program properties [12, 18, 27]. As suggested in these papers a verification problem is first translated into a set of Horn clauses with constraints in a suitable domain (usually, Linear Arithmetic), and then the satisfiability of that set of clauses is verified by using an SMT solver for Horn clauses, called here a CHC solver, such as Z3 [15] or Eldarica [19].

* This work has been partially funded by INdAM-GNCS (Italy). E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at IASI-CNR, Rome, Italy.

The main difficulty encountered by CHC solvers when verifying relational properties is that these solvers find models of *single* predicates expressed in terms of Linear Arithmetic constraints, whereas the proof of relational properties often requires the discovery of relations among arguments of *two* (*or more*) distinct predicates. To mitigate this difficulty, *Predicate Pairing* transforms a set of clauses defining two predicates, say p and q, into a new set of clauses defining a new predicate, say r, equivalent to the conjunction of p and q [12]. Thus, when the CHC solver finds a model for the predicate r, it discovers relations among the arguments of p and q.

In the approach presented in this paper we use Predicate Pairing together with *Abstraction*, which is a technique often used in program analysis and transformation. It consists in mapping the concrete semantics of a program into an abstract domain, where some program properties can more easily be verified [6]. In the context of relational verification, Predicate Pairing combined with a basic form of abstraction has been introduced in a previous paper [12]. In that paper, in fact, Predicate Pairing is performed by introducing new definitions whose bodies are made out of two atoms together with equalities between some arguments of these predicates, and these equality constraints can be viewed as an abstraction into the domain of equalities.

Abstraction is also used by CHC *Specialization*, which is another transformation technique that has been proposed to increase the effectiveness of CHC solvers [8, 20]. Given a set of clauses, CHC Specialization propagates constraints through the clauses, and since this propagation often causes strengthening of the constraints, it may be the case that, if we first specialize a given set of clauses, the task of CHC solving is made easier. However, the impact of the specialization process very much depends on the choices of the particular *abstract domain* and associated *widening operator*, which are used when the specialized predicates are introduced or manipulated.

In this paper we address the problem of evaluating various combinations of (i) Predicate Pairing, (ii) Abstraction, and (iii) Specialization for the specific objective of verifying relational properties of programs. In order to do so, we have introduced a general algorithm for Predicate Pairing that is parametric with respect to the abstract constraint domain that is used. This domain is taken to be a subdomain of Linear Arithmetic, such as *Convex Polyhedra*, *Boxes*, *Bounded Differences*, and *Octagons* [2, 3, 7, 26]. Our parametric Abstraction-based Predicate Pairing algorithm, called the APP strategy, generalizes the one that makes use of equalities between variables that has been used in a previous paper of ours [12]. We have also considered a CHC Specialization algorithm, called the ASp strategy, that is parametric with respect to the abstract constraint domain that is used, and can be viewed as a particular instance of the APP strategy. Finally, we have performed various sets of experiments by applying different sequences of the APP and ASp strategies to sets of CHCs encoding relational properties of imperative programs. In these experiments we have varied the abstract constraint domains that the strategies use and we have explored the relative merits of these different domains when verifying relational properties.

The lesson we learned from our experiments is that the strategies achieving the best results use constraint domains, such as Bounded Differences or Octagons, in which one can express relations between variables, without requiring more precise domains, such as Convex Polyhedra. Moreover, Abstraction-based Predicate Pairing essentially incorporates the effect of CHC Specialization, and thus extra specializations steps (before or after Abstraction-based Predicate Pairing) are not cost-effective.

The paper is organized as follows. In Section 2 we present an introductory example showing the usefulness of abstraction. Then, in Section 3 we present the various abstract constraint domains, such as Convex Polyhedra, Boxes, Bounded Differences, and Octagons, and the operations defined on them. In Section 4 we present the APP and ASp strategies, and we prove that they preserve satisfiability (and unsatisfiability). In Section 5 we briefly describe the implementation of our verification method based on: (i) the VeriMAP transformation and verification system, (ii) the Parma Polyhedra Library for constraint manipulation [3], and (iii) the Z3 solver for CHC satisfiability. We also report on the experiments we performed on more that one hundred verification problems. Finally, in Section 6, we discuss the related work on program transformation and verification.

2 An Introductory Example

In this section we present our running example concerning the problem of proving the equivalence of two imperative programs. CHC solvers, like Z3 [15], which are based on Linear Arithmetic are not able to prove that equivalence starting from its direct encoding in CHC. However, we will show that if we pre-process that encoding by the Predicate Pairing strategy which uses a suitable abstract constraint domain, then the Z3 solver is able to make that proof.

Let us consider the programs $P1$ and $P2$ shown in Figure 2, where program $P2$ is obtained from program $P1$ by applying a compiler optimization technique, called *software pipelining*. Software pipelining takes as input a program with a loop and produces in output a program with a new loop whose instructions are taken from different iterations of the original loop. Combined with other program transformations, software pipelining may allow more parallelism during program execution, and indeed, it can produce loops whose instructions have no read/write dependencies and thus can be executed in parallel. For example, in program $P2$, derived by pipelining from program $P1$, the dependency on `x` in the instructions of the loop in $P2$ can be removed by: (i) introducing a fresh variable `u` initialized to `x`, and (ii) replacing `x` by `u` on the right-hand side of the assignments within the loop. After this replacement we get the instructions '`u = x; y = y+u; a = a+1; x = u+a`', and we can safely execute in parallel the instruction '`y = y+u`' and the sequence of instructions '`a = a+1; x = u+a`'.

The equivalence of programs $P1$ and $P2$ with respect to the output value of `x`, can be expressed by the following clause F:

$$F\colon\ \mathit{false} \leftarrow X1 \neq X2,\ \mathit{whl}1(A, B, X, Y, A1, B1, X1, Y1),\ \mathit{ifte}(A, B, X, Y, A2, B2, X2, Y2)$$

$P1$:

```
while (a < b) {
   x = x+a;
   y = y+x;
   a = a+1;
}
```

$P2$:

```
if (a < b) {
   x = x+a;
   while (a < b-1) {
      y = y+x;
      a = a+1;
      x = x+a;
   }
   y = y+x;
   a = a+1;
}
```

Fig. 1. The input program $P1$ and the output program $P2$ obtained from $P1$ by applying software pipelining.

where: (i) predicates *whl*1 and *ifte* represent the input/output relation of programs $P1$ and $P2$, respectively, (ii) A, B, X, Y and $A1, B1, X1, Y1$ represent the values of the variables `a`, `b`, `x`, `y` at the beginning and at the end, respectively, of the execution of program $P1$, and similarly, (iii) A, B, X, Y and $A2, B2, X2, Y2$ represent the values of `a`, `b`, `x`, `y` at the beginning and at the end, respectively, of the execution of program $P2$. The clauses defining *whl*1 and *ifte*, as well as the predicate *whl*2 on which *ifte* depends, are reported below. (The non-expert reader may find the description of the technique for constructing clauses starting from imperative programs in a previous paper of ours [13].) Note that predicates *whl*1 and *whl*2 correspond to the while-loops of programs $P1$ and $P2$, respectively. Note also that strict inequalities occurring in programs (such as `a<b` in program $P1$) are represented by using non-strict inequalities in clauses (see, for instance, $A \leq B-1$ in clause 2).

1. $\mathit{whl}1(A, B, X, Y, A, B, X, Y) \leftarrow A \geq B$
2. $\mathit{whl}1(A, B, X, Y, A2, B2, X2, Y2) \leftarrow A \leq B-1, A1{=}A+1, X1 = X+A,$
   $Y1{=}X1+X, \mathit{whl}1(A1, B, X1, Y1, A2, B2, X2, Y2)$
3. $\mathit{ifte}(A, B, X, Y, A, B, X, Y) \leftarrow A \geq B$
4. $\mathit{ifte}(A, B, X, Y, A2, B2, X2, Y2) \leftarrow A \leq B-1, X1 = X+A,$
   $\mathit{whl}2(A, B, X1, Y, A2, B2, X2, Y2)$
5. $\mathit{whl}2(A, B, X, Y, A2, B, X, Y2) \leftarrow A \geq B-1, A2{=}A+1, Y2 = Y+X$
6. $\mathit{whl}2(A, B, X, Y, A2, B2, X2, Y2) \leftarrow A \leq B-2,\ A1{=}A+1,\ X1 = A1+1,$
   $Y1{=}Y+X, \mathit{whl}2(A1, B, X1, Y1, A2, B2, X2, Y2)$

Let P be the set of clauses $\{1, \ldots, 6\}$. By proving the satisfiability of $P \cup \{F\}$, we prove that programs $P1$ and $P2$ produce identical values for `x` as output, when provided with the same input values. Unfortunately, CHC solvers, like Z3, based on Linear Arithmetic cannot prove the satisfiability of $P \cup \{F\}$. This inability is due to the fact that the solver computes models of *single* predicates expressed in terms of *linear* constraints among their arguments, while *non-linear* constraints among the arguments of each predicate *whl*1 and *ifte* need be discovered to prove that the conjunction of the two atoms in the body of clause F implies $X1{=}X2$. In particular, the solver has to discover that the *whl*1 and *ifte* atoms imply $X1{=}X{+}(B^2{-}A^2{-}B{+}A)/2$ and $X1{=}X{+}(B^2{-}A^2{-}B{+}A)/2$, respectively.

The Predicate Pairing strategy we have introduced in a previous paper [12] may help in overcoming this difficulty. By Predicate Pairing we may introduce new predicates defined in terms of two (or more) atoms, together with suitable

linear constraints among their arguments. Then, CHC solvers based on Linear Arithmetic may be able to infer relations among arguments of the new predicates that correspond to conjunctions of predicates before Predicate Pairing.

However, the efficacy of the Predicate Pairing strategy crucially depends on the choice of the constraints that are added when introducing new predicates. The original Predicate Pairing strategy [12] adds equalities between arguments. In Section 4 we extend that strategy so as to be parametric with respect to the domain of constraints used, and in Section 5 we evaluate in an experimental way the effect of varying the choice of that domain for relational verification.

In Section 4, after presenting our extended Predicate Pairing transformation strategy, we complete our running example and we show that the transformation strategy that uses the constraint domain of Bounded Differences is able to prove the equivalence property.

3 Constrained Horn Clauses over Numerical Domains

Let us first recall the basic notions about: (i) some abstract domains used in static program analysis based on abstract interpretation [6], and (ii) constrained Horn clauses (CHCs).

We consider the abstract constraint domain of *Convex* (*Closed*) *Polyhedra* [2, 3, 7, 26], CP for short, over the n-dimensional real space $\mathbb{R}^n$. The *atomic constraints* of the CP domain are of the form $a_1\, x_1 + \ldots + a_n\, x_n \leq a$, where the a_i's are coefficients in $\mathbb{R}$ and the x's are variables ranging over $\mathbb{R}$. A *constraint* c is either *true*, or *false*, or an atomic constraint, or a conjunction of constraints.

Given a formula F, by $\forall(F)$ and $\exists(F)$ we denote its universal and existential closure, respectively. By $\mathit{vars}(F)$ we denote the set of variables occurring in F. A constraint c is said to be *satisfiable* if CP $\models \exists(c)$. Given two constraints c and d, we say that c *entails* d, and we write $c \sqsubseteq d$, if CP $\models \forall(c \rightarrow d)$. We say that c and d are *equivalent* if $c \sqsubseteq d$ and $d \sqsubseteq c$.

We also consider the following abstract constraint domains, namely: (i) *Univ*, (ii) *Boxes*, (iii) *Bounded Differences*, and (iv) *Octagons*, which are all subdomains of *Convex Polyhedra* in the sense that they are defined by putting restrictions on the form of the polyhedra associated with the atomic constraints. These abstract domains have all *true* and *false* as constraints and are closed under conjunction.

The constraints of the domain Univ are *true* and *false* only, and in the n-dimensional case *true* denotes the whole space $\mathbb{R}^n$ and *false* denotes the empty set on n-tuples of reals. The atomic constraints of Boxes are inequalities of the form $x \leq a$, where $a \in \mathbb{R}$. The atomic constraints of Bounded Differences are inequalities of the form $a_1\, x_1 + a_2\, x_2 \leq a$, where $a \in \mathbb{R}$, $a_i \in \{-1, 0, 1\}$, for $i = 1, 2$, and a_1 is different from a_2. The atomic constraints of Octagons are inequalities of the form $a_1\, x_1 + a_2\, x_2 \leq a$, where $a \in \mathbb{R}$ and $a_i \in \{-1, 0, 1\}$, for $i = 1, 2$.

Each abstract constraint domain $D \subseteq$ CP is endowed with some operators that we now define. (Details and examples of these operators can be found in [2, 3, 7, 26].) Let c and d be two constraints in D, or D-constraints.

The *least upper bound* operator is a function $\sqcup : D \times D \rightarrow D$ such that (i) $c \sqsubseteq c \sqcup d$, (ii) $d \sqsubseteq c \sqcup d$ and (iii) for all D-constraints e, if $c \sqsubseteq e$ and $d \sqsubseteq e$, then $c \sqcup d \sqsubseteq e$.

A *widening operator* is a function $\nabla : D \times D \rightarrow D$ such that (i) $c \sqsubseteq c \nabla d$, (ii) $d \sqsubseteq c \nabla d$, and (iii) for all chains $y_0 \sqsubseteq y_1 \sqsubseteq \ldots$, the chain $x_0 \sqsubseteq x_1 \sqsubseteq \ldots$, where $x_0 = y_0$ and, for $i > 0$, $x_{i+1} = x_i \nabla y_{i+1}$, has finitely many distinct elements (modulo equivalence in Linear Arithmetic).

The *abstraction operator* for a subdomain D of CP, is a function $\alpha : \mathrm{CP} \rightarrow D$ such that (i) $c \sqsubseteq \alpha(c)$, and (ii) for all D-constraints e, if $c \sqsubseteq e$, then $\alpha(c) \sqsubseteq e$.

The *projection* of a D-constraint c onto a set X of variables, denoted $c \Downarrow X$, is a D-constraint c', with variables in X, which is equivalent to $\exists Y.c$, where $Y = vars(c) - X$. Clearly, $c \sqsubseteq c'$.

An *atom* is a formula of the form $p(X_1, \ldots, X_m)$, where p is a predicate symbol different from '$\leq$' and $X_1, \ldots, X_m$ are distinct variables. A *constrained Horn clause* (or simply, a *clause*, or a CHC) is an implication of the form $A \leftarrow c, G$ (comma denotes conjunction), where the conclusion (or *head*) A is either an atom or *false*, the premise (or *body*) is the conjunction of a constraint c and a (possibly empty) conjunction G of atoms. The empty conjunction is identified with *true*. We also assume that two atoms in the body of a clause do not share any variable. Note that, for reasons of simplicity, we wrote the clauses of the example in Section 2 in a form which does not comply with the syntax defined in this section. However, they can be rewritten into a compliant form by applying the following transformations: (i) the removal of multiple occurrences of variables in (conjunctions of) atoms in favor of equalities, (ii) the replacement of equalities by conjunctions of inequalities, and (iii) the split of clause F into two clauses where the disequality $X1 \neq X2$ has been replaced by the inequalities $X1 \leq X2 - 1$ and $X1 \geq X2 + 1$, respectively.

A set S of CHCs is said to be *satisfiable* if $S \cup \mathrm{CP}$ has a model, or equivalently, $S \cup \mathrm{CP} \not\models \mathit{false}$.

4 Predicate Pairing with Abstraction

In this section we present an algorithm for transforming CHCs, called *Abstraction-based Predicate Pairing* (or *APP strategy*, for short), which combines Predicate Pairing [12] with abstraction operators acting on a given constraint domain (see Figure 2). The APP transformation strategy preserves satisfiability of clauses and has the objective of increasing the effectiveness of the satisfiability check that is performed by the subsequent application of a CHC solver.

The APP transformation strategy tuples together two or more predicates into a single new predicate which is equivalent to their conjunction. As discussed in Section 2, the addition of suitable constraints among the variables of the predicates paired together (or tupled together, if more than two), may ease the discovery of the relations existing among the arguments of the individual predicates. The APP strategy is parametric with respect to: (i) the abstract constraint domain which is considered, and (ii) a *Partition* operator that determines, for a given a clause, the atoms to be tupled together by splitting the

conjunction G of atoms in the body of the clause into n (≥ 1) subconjunctions $G_1, \ldots, G_n$. By choosing the abstract constraint domain and the Partition operator in suitable ways, we can derive a wide range of transformations, and among these, the Predicate Pairing strategy introduced in a previous paper [12], *Linearization* [10], and CHC Specialization [8, 20].

In particular, the Predicate Pairing strategy is derived as follows. The constraint domain is the set of equalities between variables (thus, a subdomain of the Bounded Differences domain). For defining the Partition operator, suppose that the goal of the strategy is to pair two predicates q and r defined by two disjoint sets of clauses Q and R, respectively. Then, for a clause $H \leftarrow c, Q_1, \ldots, Q_m, R_1, \ldots, R_n$, with $m \leq n$, where $Q_1, \ldots, Q_m$ are atoms defined by clauses in Q and $R_1, \ldots, R_n$ are atoms defined by clauses in R, the Partition operator returns the partition $(Q_1, R_1), \ldots, (Q_m, R_m), (R_{m+1}), \ldots, (R_n)$, and similarly for the case $m \geq n$ (more sophisticated ways of choosing (Q_i, R_j) pairs have been proposed [14]).

A CHC Specialization strategy with Abstraction, which we call *ASp strategy*, can be derived by instantiating the APP strategy as we now specify. The ASp strategy is obtained by using the Partition operator that, given a conjunction of atoms $A_1, \ldots, A_n$ in the body of a clause, returns n subconjunctions, each consisting of a single atom A_i, with $i \in \{1, \ldots, n\}$ (that is, Predicate Pairing is not performed). In Section 5 we will show the effects of using ASp, together with APP, with different abstract constraint domains.

The APP strategy is realized by performing a sequence of applications of the well-known *unfold/fold rules* [17]. In order to be self-contained, now we present the version of the Unfolding rule used in this paper. The other rules will be presented when describing the APP strategy.

Unfolding Rule. Let P be a set of clauses and C be a clause of the form $H \leftarrow c, L, A, R$, where A is an atom and L and R are (possibly empty) conjunctions of atoms. Let us consider the set $\{A \leftarrow c_i, B_i \mid i = 1, \ldots, m\}$ made out of all the clauses in P whose head is A (after renaming). By unfolding C w.r.t. A using P, we derive the set of clauses $\{(H \leftarrow c, c_i, L, B_i, R) \mid i = 1, \ldots, m\}$.

The APP strategy constructs a tree *Defs* of clauses whose head is either *false* or a new predicate, that is, a predicate not occurring in the input set P of clauses. Clauses with new head predicates are called *definitions*. A definition D is said to be a *child* of a definition C, and equivalently, C is said to be the *parent* of D, if D is introduced to fold a clause derived by unfolding from clause C. The *ancestor* relation on *Defs* is the *reflexive transitive* closure of the parent relation.

Note that, by construction, every constraint, either a_i or d_i, occurring in a new definition D_i introduced during the DEFINITION & FOLDING phase (see Figure 2), belongs to the abstract constraint domain.

Let us now prove the termination and soundness of the strategy. A Partition operator is said to be *bounded* if there exists a positive integer k such that, for any clause C, the operator splits the body of C into the subconjunctions $G_1, \ldots, G_n$, where, for $i = 1, \ldots, n$, the number of atoms in G_i is at most k. For instance, $k \leq 2$ is a bound for the Partition operators described above.

Input: A set $P \cup \{C\}$ of clauses where C is a clause whose head is *false*.
Output: A set *TransfCls* of clauses.

INITIALIZATION: $InCls := \{C\}$; *Defs* is the tree made out of the root clause C only; $TransfCls := P$;

while there is a clause C in *InCls* of the form $H \leftarrow c, B$ *do*

- UNFOLDING: From clause C derive a set $U(C)$ of clauses by unfolding C with respect to each atom occurring in its body using P;
- CLAUSE DELETION: Remove from $U(C)$ all clauses with an unsatisfiable constraint;
- DEFINITION & FOLDING:
  for every clause $E \in U(C)$ of the form $H \leftarrow d, G$ *do*
    Partition the conjunction G into $n\,(\geq 1)$ subconjunctions $G_1, \ldots, G_n$;
    for $i = 1, \ldots, n$ *do*
      $d_i := \alpha(d) \Downarrow V_i$, where V_i is the set of variables in G_i;
      if in *Defs* there is no clause $newp_i(V_i) \leftarrow e_i, G_i$ such that $d_i \sqsubseteq e_i$ *then*
        if in *Defs* there is an ancestor clause of C of the form $newq(V_i) \leftarrow f_i, G_i$
        then $D_i := (newp_i(V_i) \leftarrow a_i, G_i)$, where $a_i = f_i \nabla (f_i \sqcup d_i)$
        else $D_i := (newp_i(V_i) \leftarrow d_i, G_i)$;
        $InCls := InCls \cup \{D_i\}$; add D_i as a child of C in *Defs*;
    end-for;
    $TransfCls := TransfCls \cup \{H \leftarrow d, newp_1(V_1), \ldots, newp_n(V_n)\}$;
  end-for;
  $InCls := InCls - \{C\}$;

end-while

Fig. 2. The APP transformation strategy.

Theorem 1 (Termination and Soundness of Predicate Pairing with Abstraction). *Let the set $P \cup \{C\}$ of clauses be the input of the APP strategy. Suppose that APP uses a bounded Partition operator. Then, the strategy terminates and returns a set TransfCls of clauses such that $P \cup \{C\}$ is satisfiable iff TransfCls is satisfiable.*

Proof. (Sketch) Since the Partition operator is bounded and, by definition, no sequence of applications of the widening operator ∇ can generate infinitely many distinct constraints (modulo equivalence), the set of new predicate definitions that can be introduced by the APP strategy is finite. Thus, the number of executions of the *while* loop of the strategy is also finite, and hence APP terminates.

To show the soundness of APP we first recall the following result (see Theorem 2 in [12], which is a consequence of a well-known result by Etalle and Gabbrielli [17]): Suppose that from a set *Cls* of clauses we derive a new set *TransfCls* of clauses by a sequence of applications of the unfold/fold rules, such that every definition used for folding is unfolded during that sequence. Then *Cls* is satisfiable iff *TransfCls* is satisfiable. Now, by taking *Cls* to be the set $P \cup \{C\}$ of clauses that are an input of APP, the thesis follows from the fact that every clause added to *Defs* (and hence to *InCls*) is eventually unfolded. □

Let us see the APP strategy in action on the example of Section 2. As already mentioned, the disequality $X1 \neq X2$ is viewed as a disjunction of two inequalities,

and hence clause F is split into two clauses, say $F1$ and $F2$, containing the two disjuncts. For reasons of space we only prove the satisfiability of the set $P \cup \{F1\}$, where $F1$ is the following clause:

$F1\colon\ \mathit{false} \leftarrow X1 \leq X2-1,\ \mathit{whl}1(A,B,X,Y,A1,B1,X1,Y1),$
$\qquad\qquad\qquad\qquad\qquad\qquad \mathit{ifte}(A,B,X,Y,A2,B2,X2,Y2)$

The satisfiability of $P \cup \{F1, F2\}$ can be proved by applying the strategy twice. For the application of the APP strategy we use the Bounded Differences domain, or *BDS*, for short.

After the Initialization step, the APP strategy selects $F1$ from *InCls* and applies the Unfolding step. The unfolding of *whl*1 and *ifte* occurring in the body of $F1$ mimics the execution of the *while* loop of program P1 and the *if-then-else* of program P2. By unfolding we get four clauses, three of which have unsatisfiable constraints and are removed by the subsequent Clause Deletion step. The only clause with a satisfiable constraint is the following one (up to equivalence in Linear Arithmetic and variable renaming):

7. $\mathit{false} \leftarrow X2 \leq X4-1, A1 \leq B1, A1 = A3+1, Y1 = Y3+X1, B1 = B3, X1 = X3,$
$\qquad \mathit{whl}1(A1,B1,X1,Y1,A2,B2,X2,Y2),$
$\qquad \mathit{whl}2(A3,B3,X3,Y3,A4,B4,X4,Y4).$

Then the Definition & Folding step adds the following new definition to *Defs* and to *InCls*:

8. $\mathit{pp}(A1,B1,X1,Y1,A2,B2,X2,Y2,A3,B3,X3,Y3,A4,B4,X4,Y4) \leftarrow$
$\quad X2 \leq X4-1,\ A1 \leq B1,\ A1 = A3+1,\ B1 = B3,\ X1 = X3,$
$\quad \mathit{whl}1(A1,B1,X1,Y1,A2,B2,X2,Y2), \mathit{whl}2(A3,B3,X3,Y3,A4,B4,X4,Y4).$

The body of clause 8 consists of the conjunction of the two atoms occurring in the body of clause 7 together with the constraints obtained from the constraints of clause 7 by applying the abstraction operator α for BDS (the projection is the identity in this case, because the variables occurring in the constraints are a subset of the variables in the atoms). The operator α drops the constraint $Y1 = Y3+X1$ whose least overapproximation in BDS is *true*. By folding clause 7 using definition 8 we get the following clause, which is then added to *TranfCls*:

9. $\mathit{false} \leftarrow X2 \leq X4-1, A1 \leq B1, A1 = A3+1, Y1 = Y3+X1, B1 = B3, X1 = X3,$
$\qquad \mathit{pp}(A1,B1,X1,Y1,A2,B2,X2,Y2,A3,B3,X3,Y3,A4,B4,X4,Y4).$

Now, the APP strategy performs a second iteration of the *while* loop to process definition 8 in *InDefs*. By Unfolding and Clause Deletion we have that the constraints occurring in definition 8 are preserved. Hence, we get a clause that can be folded again using definition 8, thereby deriving:

10. $\mathit{pp}(A1,B1,X1,Y1,A2,B2,X2,Y2,A3,B3,X3,Y3,A4,B4,X4,Y4) \leftarrow$
$\qquad X2 \leq X4-1,\ A1 \leq B1,\ A1 = A3+1,\ B1 = B3,\ X1 = X3,\ A5 \leq B1,$
$\qquad A5 = A1+1,\ X5 = X1+A1,\ Y5 = Y1+X5,\ X6 = X5,\ Y6 = Y3+X1,$
$\qquad \mathit{pp}(A5,B1,X5,Y5,A2,B2,X2,Y2,A1,B3,X6,Y6,A4,B4,X4,Y4).$

Since there are no more clauses to be processed in *InCls*, the final set of clauses is *TransfCls*=$\{9,10\}$. The satisfiability of *TransfCls* is trivial, and is easily checked

by Z3, because it contains no constrained facts (that is, clauses with only constraints in their body), and hence a model is obtained by taking *pp* to be *false*.

Note that the widening and least upper bound operators were not used in our running example, but widening is needed, in general, to guarantee the termination of APP (see Theorem 1).

Note also that the abstract constraint domain used by APP is crucial for deriving clauses without constrained facts. Indeed, if in our running example we use the domain Univ, instead of BDS, then the body of the new definition introduced after unfolding consists of the conjunction of the two *whl*1 and *whl*2 atoms without any constraint, as the abstraction operator for Univ maps every satisfiable constraint to *true*. Then, by unfolding this new definition, we get a constrained fact derived from the constrained facts of *whl*1 and *whl*2 (i.e., clauses 1 and 5 of Section 2).

5 Experimental Evaluation

In this section we present the results of the experiments we have performed by applying in various ways both the APP strategy and its instance, the ASp strategy. These results illustrate the role of abstract constraint domains considered when applying those strategies, and they also show the usefulness of the APP strategy for improving the performance of the CHC solvers when checking satisfiability of clauses.

We have implemented the APP and the ASp strategies using the VeriMAP transformation system [9] and the Parma Polyhedra Library (PPL) [3], and we have used the Z3 solver [15] for checking satisfiability of the clauses generated by those strategies. The verification process is depicted in Figure 3 and can be described as follows. The clauses encoding a verification problem are given as input to the VeriMAP system which applies to them a (possibly empty) sequence of APP (or ASp) strategies, using a specific constraint domain. When applying these strategies the constraints in the domain abstract are manipulated using the Parma Polyhedra Library. The resulting clauses, if produced within a specified timeout, will be passed in input to the Z3 solver to test their satisfiability.

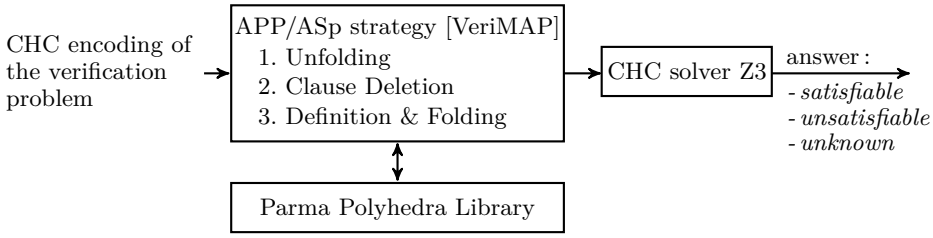

Fig. 3. The verification process.

Implementation of the APP and ASp strategies. We have ported the VeriMAP system from SICStus Prolog 3.12.5 to SWI-Prolog 7.4.2 and we have extended

its transformation engine so to use the abstract constraint domains and the associated operations provided by the Parma Polyhedra Library 1.2.

The domains we have considered are: (i) *Univ*, (ii) *Boxes*, (iii) *Bounded Differences* (also called Bounded Difference Shapes, and denoted BDS, for short), (iv) *Octagons* (also called Octagonal Shapes, and denoted OS, for short), and (v) *Convex Polyhedra*, together with the operations of projection, least upper bound, widening, emptiness check, and inclusion check (these two kinds of checks correspond to satisfiability and entailment, respectively). In particular, we have considered the following two variants of the Convex Polyhedra domain: (1) the one with the widening operator of Halbwachs [7], and (2) the one with the widening operator of Bagnara et al. [2]. These variants will be denoted by CP-H and CP-B, respectively. Since VeriMAP natively represents constraints using the syntax of the Constraint Logic Programming (CLP), when implementing the APP strategy, we have used the translation from PPL polyhedra to CLP constraints, and vice versa.

Benchmark suite. We have considered a benchmark suite consisting of 136 verification problems, for a total number of 1655 input constrained Horn clauses. Each problem consists in the verification of a relational property, such as equivalence, monotonicity, injectivity, functional dependency, loop optimization, and non-interference [4, 5, 10, 12, 18].

Experiments. We have performed the following six sets of experiments E0–E5 (see the corresponding six frames in Table 1):

E0: Z3
E1: ASp(X) ; Z3
E2: ASp(X) ; APP(X) ; Z3
E3: ASp(X) ; APP(X) ; ASp(X) ; Z3
E4: APP(X) ; Z3
E5: APP(Univ) ; ASp(X) ; Z3

where: (i) the parameter X is an abstract domain in the set {Boxes, BDS, OS, CP-H, CP-B} and (ii) ASp(X) and APP(X) denote an application of the Abstraction-based Specialization strategy and the Abstraction-based Predicate Pairing strategy, respectively, by using the abstract domain X.

When trying to solve a single verification problem, we set a timeout of 300 seconds for each application of the APP(X) strategy or of the ASp(X) strategy or of the Z3 solver.

Here is an explanation of the experiments E0, E1 with X=OS, and E2 with X=OS. The explanation of the other experiments is similar.

Experiment E0 (see Frame E0 in Table 1) consists in performing a run of Z3 directly on the clauses that encode each verification problem. Z3 solves (either positively or negatively) 28 problems (see Column *SolProbls*) out of the total 136 verification problems, by providing the answer (either '*satisfiable*' or '*unsatisfiable*', respectively) within the timeout in an average time of 2.36 seconds (see Column *AvgTime*2) per solved problem.

Experiment E1 with X = OS (see line OS of Frame E1 in Table 1) consists in applying the ASp(OS) strategy on the clauses that encode each of the 136

	VeriMAP					Z3	
Exp	*Domain* X	*OutProbls*	*OutCls*	*SizeRatio*	*AvgTime*1	*SolProbls*	*AvgTime*2
E0	—	—	—	—	—	28	2.36
E1	Boxes	136	3111	1.88	0.67	29	3.15
	BDS	136	2629	1.59	0.66	28	3.79
	OS	136	3540	2.14	0.73	28	4.10
	CP-H	136	3021	1.83	0.66	34	3.95
	CP-B	136	3633	2.20	0.69	36	10.14
E2	Boxes	134	27753	17.10	2.52	73	2.20
	BDS	136	12793	7.73	3.26	119	3.69
	OS	134	20361	12.44	5.23	121	3.90
	CP-H	135	16193	9.84	3.74	113	0.93
	CP-B	127	12554	8.06	3.51	114	3.65
E3	Boxes	134	45970	28.32	5.09	77	3.54
	BDS	136	26683	16.12	6.56	121	3.86
	OS	134	36871	22.52	10.21	119	3.06
	CP-H	135	31521	19.16	7.66	115	2.05
	CP-B	127	25495	16.37	8.10	112	1.27
E4	Boxes	136	20296	12.26	2.27	78	2.01
	BDS	136	8630	5.21	1.38	121	2.45
	OS	135	13762	8.37	2.97	120	1.77
	CP-H	135	13823	8.40	2.59	110	1.57
	CP-B	131	11718	7.35	2.22	113	2.19
E5	Boxes	136	19932	12.04	2.94	74	3.07
	BDS	136	8387	5.07	2.17	120	1.63
	OS	135	14065	8.55	3.64	118	1.39
	CP-H	135	14111	8.58	3.29	112	1.44
	CP-B	129	9831	6.24	3.12	113	2.05

Table 1. Column *Exp* reports the set of experiments considered in each frame. Every line in each frame reports the results of a single experiment which consists of 136 verification problems. The abstract domain used in an experiment is shown in Column *Domain* X. Columns *OutProbls* and *OutCls* report the number of non-aborted verification problems and the total number of their output clauses, respectively. Column *SizeRatio* reports the value *OutCls* divided by the total number of input clauses of the non-aborted verification problems. Column *AvgTime*1 reports the time taken to produce the clauses of Column *OutCls* divided by the value *OutProbls*. Columns *SolProbls* and *AvgTime*2 report the number of (non-aborted) verification problems solved by Z3 and the average time taken by Z3 per solved problem. The times are the CPU seconds spent in user mode.

verification problems, and then running Z3. From a total of 1655 input clauses these 136 applications of ASp(OS) produce a total of 3540 output clauses (see Column *OutCls*) with a size increase of about 2.14 ($\approx$3540/1655) times (see Column *SizeRatio*), in an average time of 0.73 seconds per problem (see Column *AvgTime*1). Then, on the 3540 output clauses we run Z3 that solves 28 problems with an average time of 4.10 seconds per solved problem.

Experiment E2 with X = OS (see line OS of Frame E2 in Table 1) consists in applying the ASp(OS) strategy on the input clauses, exactly as in Experi-

ment E1, then applying the APP(OS) strategy, which produces a total of 20361 output clauses, and finally running Z3. Note that for two problems APP(OS) is unable to produce the output clauses within the timeout (see Column *OutProbls* where the entry is 134, instead of 136).

When trying to solve a verification problem among the 136 problems of a given experiment, it may be the case that the ASp(X) strategy, or the APP(X) strategy, does not complete its execution within the timeout. In that case we say that the verification problem is *aborted* and the input clauses encoding that problem are not taken into account when computing the size ratios of Column *SizeRatio*. Similarly, the time taken for any aborted verification problem is not taken into account when computing the average times of Column *AvgTime*1.

In all our experiments we have used as constraint solver Z3 4.5.0 with the Duality fixed-point engine [24] on an Intel Xeon CPU E5-2640 2.00GHz processor with 64GB of memory under the GNU/Linux 64 bit operating system CentOS 7.

Discussion of the Results. Let us now comment on the experimental results presented in Table 1. First we observe that various combinations of the ASp and APP strategies (or the APP strategy alone) significantly increase the number of problems that Z3 solves. Indeed, while Z3 alone solves 28 problems only (see Frame E0), suitable combinations of the ASp and APP strategies (or APP alone) allow Z3 to solve over 120 problems (see Frames E2–E5).

However, the increase of efficacy in proving the desired properties is mainly due to the APP strategy, rather then the ASp strategy. Indeed, Frame E1 shows that the use of ASp alone makes just a marginal increase in the number of problems solved by Z3 (from the 28 solved problems, as shown in Frame E0, to a maximum of 36 solved problems, as shown in Frame E1). Moreover, by combining the ASp and APP strategies (see Frames E2 and E3, Column *SolProbls*) we get results which are not significantly better than the ones obtained by using the APP strategy alone (see Frame E4, Column *SolProbls*).

The comparison between Frames E4 and E5 (Columns for Z3) tells us that the effect of the APP(X) strategy, for a given abstract domain X, can also be obtained in two steps: (i) first, by applying APP(Univ), and (ii) then, by applying ASp(X). Recall that the abstraction operator for the Univ domain maps any satisfiable constraint to *true*, and hence APP(Univ) does not add any constraint when new definitions are introduced. In other terms, APP(Univ); ASp(X) separates Predicate Pairing from constraint addition using domain X, whereas APP(X) does the two transformations at the same time.

Let us now analyze our results from the perspective of the constraint domain X used in ASp(X) and APP(X). The use of the Boxes domain, that is, interval constraints on single real variables, is not very effective. Indeed, in Frames E2–E5 we see that Boxes allows the solution of at most 78 problems (see Frame E4, line 'Boxes'), while the other domains enable Z3 to solve at least 110 problems (see Frame E4, line 'CP-H'). The poor performance of Boxes with respect to those of the other domains can be explained by the fact that constraints in Boxes are not expressive enough to represent relations among program variables. Hence, they are of little help for proving relational properties.

Note, however, that if precision is increased, from BDS and OS to Convex Polyhedra (CP-H or CP-B), the efficacy of the verification process decreases. For instance, Frame E4 shows that APP(BDS) and APP(OS) allow Z3 to solve 121 and 120 problems, respectively, while APP(CP-H) and APP(CP-B) allow Z3 to solve at most 113 problems. We would also like to point out that the sets of problems solved with two different abstract domains are not always comparable. For instance, in Frame E2 the set of 119 problems solved with BDS is not a subset of the 121 problems solved with OS. In particular, two problems were solved by using BDS and not by using OS.

Finally, we would like to comment on the computational performances of the transformations. Some combinations of ASp and APP significantly increase the number of output clauses (see, in particular, the increase of over 28 times shown in Frame E3, line 'Boxes', Column *SizeRatio*) and are costly (see, for instance, the average time of 10.21 seconds in Frame E3, line 'OS', Column $AvgTime1$). This is mainly due to the fact that ASp may introduce several specialized versions for the same predicate occurring in the original set of clauses. However, if we consider the APP(BDS) strategy, without previous or subsequent applications of ASp (see Frame E4), then the increase of the number of output clauses is limited to about 5 times and the average transformation time is only 1.38 seconds, and hence much lower than the average solving time taken by Z3.

6 Related Work and Conclusions

We have proposed various ways of combining transformation and abstraction techniques for constrained Horn clauses with the goal of verifying relational properties of imperative programs. To this aim we have presented two algorithms, the Predicate Pairing and Specialization algorithms, which are parameterized with respect to a given abstract constraint domain and its operators. Then we have presented an extensive experimental evaluation of CHC satisfiability problems encoding relational verification problems. Our experiments show that suitable combinations of transformations and abstraction dramatically increase the effectiveness of the Z3 solver on the given benchmark. The most effective techniques combine Predicate Pairing and Abstraction based on Bounded Differences or Octagons [2, 26], that is, constraint domains that are quite simple, but expressive enough to capture the relations between predicate arguments.

Relational verification has been extensively studied, and still receives much attention as a relevant problem in the field of software engineering [4, 12, 18, 22, 27]. In particular, during the software development process it may be helpful to prove that the semantics of a new program version has some specified relation with the semantics of an old version.

Among the various methods to prove relational properties, those by Mordvinov and Fedyunkovich [27] and by Felsing et al. [18] are the most closely related to ours. The method proposed in the former paper [27] introduces the notion of CHC product (somewhat related to Predicate Pairing), that is, a CHC transformation that synchronizes computations to improve the effectiveness of the CHC satisfiability checks. The latter method proposed by Felsing et al. [18] presents

proof rules for relations between imperative programs that are translated into constrained Horn clauses. The satisfiability of these clauses, which entails the relation of interest, is then checked by state-of-the-art CHC solvers.

The Predicate Pairing technique we present in this paper is a descendant of well-known techniques for logic program transformation, such as *Tupling* [29] and *Conjunctive Partial Deduction* [16], which derive new predicates defined in terms of conjunctions of atoms. The goal of these techniques is to derive efficient logic programs by: (i) avoiding multiple traversals of data structures and repeated evaluations of predicate calls, and (ii) producing specialized program versions that take into account partial information on the input values. An integration of Conjunctive Partial Deduction and abstract interpretation, called *Abstract* Conjunctive Partial Deduction, has also been presented in the literature [23]. Recent work has shown that the extension of these transformation techniques to constrained Horn clauses can play a significant role in improving the effectiveness of CHC solvers for proving properties of imperative programs, and in particular for verifying relational properties [11, 12].

The CHC Specialization strategy we consider in this paper is a variant of specialization techniques for (constraint) logic programs which have been proposed to support program verification [1, 8, 10, 13, 20, 21, 25, 28]. However, these techniques are focused on the verification of partial or total correctness of single programs, and not on the relational verification.

7 Acknowledgements

We thank the anonymous referees for their constructive comments.

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In Proc. *PADL '07*, LNCS 4354, pages 124–139. Springer, 2007.
2. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Sci. Comput. Program.*, 58(1):28–56, 2005.
3. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1–2):3–21, 2008.
4. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In Proc. *FM '11*, LNCS 6664, pages 200–214. Springer, 2011.
5. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In Proc. *POPL '04*, pages 14–25. ACM, 2004.
6. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In Proc. *POPL '77*, pages 238–252. ACM, 1977.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proc. *POPL '78*, pages 84–96. ACM, 1978.
8. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95, Part 2:149–175, 2014.
9. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. Proc. *TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014. Available at: `http://www.map.uniroma2.it/VeriMAP`.

10. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming*, 15(4-5):635–650, 2015.
11. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 140(3-4):329–355, 2015.
12. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Relational verification through Horn clause transformation. In Proc. *SAS '16*, LNCS 9837, pages 147–169. Springer, 2016.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. *Sci. Comput. Program.*, 147 (78–108), 2017.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Predicate pairing for program verification. *Theory and Practice of Logic Programming*, 1-41. doi:`https://doi.org/10.1017/S1471068417000497`, 2017.
15. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In Proc. *TACAS '08*, LNCS 4963, pages 337–340. Springer, 2008.
16. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
17. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
18. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. Proc. *ASE '14*, pages 349–360. ACM, 2014.
19. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems. In Proc. *FM '12*, LNCS 7436, pages 247–251. Springer, 2012.
20. B. Kafle and J. P. Gallagher. Constraint specialisation in Horn clause verification. *Sci.Comput.Program.*, 137:125–140, 2017.
21. B. Kafle and J. P. Gallagher. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures*, 47:2–18, 2017.
22. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In Proc. *ESEC/FSE '13*, pages 345–355. ACM, 2013.
23. M. Leuschel. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM TOPLAS*, 26(3):413–463, 2004.
24. K. L. McMillan and A. Rybalchenko. Solving constrained Horn clauses using interpolation. MSR Technical Report 2013-6, Microsoft Report, 2013.
25. M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In Proc. *LOPSTR '07*, LNCS 4915, pages 154–168. Springer, 2008.
26. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
27. D. Mordvinov and G. Fedyukovich. Synchronizing constrained Horn clauses. In Proc. *LPAR '17*, *EPiC Series in Computing* 46, pages 338–355. EasyChair, 2017.
28. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In Proc. *SAS '98*, LNCS 1503, pages 246–261. Springer, 1998.
29. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.