Predicate Pairing for Program Verification

EMANUELE DE ANGELIS, FABIO FIORAVANTI
DEC, 'G. d'Annunzio' University of Chieti-Pescara, Pescara, Italy
(*e-mail:* {emanuele.deangelis,fabio.fioravanti}@unich.it)

ALBERTO PETTOROSSI
DICII, University of Rome Tor Vergata, Rome, Italy
(*e-mail:* pettorossi@info.uniroma2.it)

MAURIZIO PROIETTI
CNR-IASI, Rome, Italy
(*e-mail:* maurizio.proietti@iasi.cnr.it)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

It is well-known that the verification of partial correctness properties of imperative programs can be reduced to the satisfiability problem for constrained Horn clauses (CHCs). However, state-of-the-art solvers for constrained Horn clauses (or CHC solvers) based on *predicate abstraction* are sometimes unable to verify satisfiability because they look for models that are definable in a given class $\mathcal{A}$ of constraints, called $\mathcal{A}$-definable models. We introduce a transformation technique, called *Predicate Pairing*, which is able, in many interesting cases, to transform a set of clauses into an equisatisfiable set whose satisfiability can be proved by finding an $\mathcal{A}$-definable model, and hence can be effectively verified by a state-of-the-art CHC solver.

In particular, we prove that, under very general conditions on $\mathcal{A}$, the unfold/fold transformation rules preserve the existence of an $\mathcal{A}$-definable model, that is, if the original clauses have an $\mathcal{A}$-definable model, then the transformed clauses have an $\mathcal{A}$-definable model. The converse does not hold in general, and we provide suitable conditions under which the transformed clauses have an $\mathcal{A}$-definable model *if and only if* the original ones have an $\mathcal{A}$-definable model. Then, we present a strategy, called Predicate Pairing, which guides the application of the transformation rules with the objective of deriving a set of clauses whose satisfiability problem can be solved by looking for $\mathcal{A}$-definable models. The Predicate Pairing strategy introduces a new predicate defined by the conjunction of two predicates occurring in the original set of clauses, together with a conjunction of constraints. We will show through some examples that an $\mathcal{A}$-definable model may exist for the new predicate even if it does not exist for its defining atomic conjuncts. We will also present some case studies showing that Predicate Pairing plays a crucial role in the verification of *relational properties of programs*, that is, properties relating two programs (such as program equivalence) or two executions of the same program (such as non-interference). Finally, we perform an experimental evaluation of the proposed techniques to assess the effectiveness of Predicate Pairing in increasing the power of CHC solving.

KEYWORDS: Program Verification, Constrained Horn Clauses, Constraint Logic Programming, Program Transformation, Relational Properties of Programs

1 Introduction

Constrained Horn clauses (CHCs, for short) have been advocated by many researchers as a suitable logical formalism for the specification and the automated verification of properties of imperative programs (Albert et al. 2007; Bjørner et al. 2015; De Angelis et al. 2014a; Jaffar et al. 2009; Kafle et al. 2016; Méndez-Lojo et al. 2008; Peralta et al. 1998; Podelski and Rybalchenko 2007; Rümmer et al. 2013). In particular, the problem of showing *partial correctness* properties defined by Hoare triples (Hoare 1969) has a natural translation into the problem of proving the *satisfiability* of a suitable set of constrained Horn clauses.

Consider, for instance, the C-like program `sum_upto` in Figure 1, which computes the sum of the first `m` non-negative integer numbers:

```
int m, sum;
int f(int x) {
   int r = 0;
   while (x > 0) {
      r = r + x;  x--; }
   return r;
   }
void sum_upto() {
   sum = f(m);
   }
```

Figure 1. Program `sum_upto` computing $\texttt{sum} = \sum_{x=1}^{\texttt{m}} x$.

Suppose we want to prove the following Hoare triple: $\{\texttt{m} \geq 0\}$ `sum_upto` $\{\texttt{sum} \geq \texttt{m}\}$. This triple is valid if the following set of clauses, called *verification conditions*, is satisfiable:

1. $\mathit{false} \leftarrow M > \mathit{Sum},\ M \geq 0,\ R = 0,\ \mathit{su}(M, R, \mathit{Sum})$
2. $\mathit{su}(X, R, \mathit{Sum}) \leftarrow X \leq 0,\ \mathit{Sum} = R$
3. $\mathit{su}(X, R, \mathit{Sum}) \leftarrow X > 0,\ R1 = R + X,\ X1 = X - 1,\ \mathit{su}(X1, R1, \mathit{Sum})$

These clauses can be obtained in an automatic way from an interpreter of the C-like imperative language we consider and the given Hoare triple by using a technique described in the literature (Albert et al. 2007; De Angelis et al. 2014a; De Angelis et al. 2017; Méndez-Lojo et al. 2008; Peralta et al. 1998). The predicate $\mathit{su}(M, R, \mathit{Sum})$, which holds iff $\mathit{Sum} = R + \sum_{x=1}^{M} x$, encodes the operational semantics of the program `sum_upto`. Clause 1 encodes the Hoare triple, stating that if $\mathit{su}(M, R, \mathit{Sum})$ holds with $M \geq 0$ (that is, the precondition $\texttt{m} \geq 0$ holds) and $R = 0$ (that is, `r` is initialized to 0), then $\mathit{Sum} \geq M$ (that is, at the end of the execution, the value of the variable `sum` is not smaller than the value of the variable `m`). Clauses 2 and 3 encode the while-loop of the function `f`.

Constrained Horn clauses are syntactically the same as *constraint logic programs* (Jaffar and Maher 1994). However, the term 'constrained Horn clauses' is mostly used in the field of program verification and, unlike 'constraint logic programs', it is not associated with any operational meaning. Moreover, most of the research on constrained Horn clauses is devoted to finding a model, expressible in the constraint theory, that proves the *satisfiability* of the clauses, whereas the operational semantics of constraint logic programs is based on a refutation procedure

that looks for a proof of the *unsatisfiability* of the clauses. In this respect, the techniques used for finding models of constrained Horn clauses are closer to the ones proposed for the static analysis of constraint logic programs based on abstract interpretation (Cousot and Cousot 1977; Benoy and King 1997), where the objective is to find an over-approximation of the least model of the program.

The proof of satisfiability of sets of constrained Horn clauses is supported by *CHC solvers* that have been developed in recent years for various constraint theories, such as (linear or nonlinear) integer arithmetic, real (or rational) arithmetic, booleans, integer arrays, lists, heaps, and other data structures (De Angelis et al. 2014b; Grebenshchikov et al. 2012; Gurfinkel et al. 2015; Hoder et al. 2011; Hojjat et al. 2012; Kafle et al. 2016; McMillan and Rybalchenko 2013). However, in general, since the satisfiability of constrained Horn clauses is an undecidable problem, CHC solvers may not be able to return conclusive answers.

In order to improve the effectiveness of CHC solvers, several techniques proposed by recent papers perform satisfiability preserving transformations on sets of clauses that, in some cases, derive clauses whose satisfiability is easier to prove (De Angelis et al. 2014a; De Angelis et al. 2015a; De Angelis et al. 2015b; De Angelis et al. 2016; Kafle and Gallagher 2017a; Kafle and Gallagher 2017b). These transformations are adaptations to the task of improving the effectiveness of satisfiability checking of earlier techniques which were developed for improving the efficiency of execution of (constraint) logic programs, such as *query answer transformation*, *specialization* (or *partial deduction*), and *unfold/fold transformations* (Debray and Ramakrishnan 1994; Etalle and Gabbrielli 1996; Leuschel and Bruynooghe 2002; Tamaki and Sato 1984; Pettorossi and Proietti 1994).

In this paper we further enhance the approach to CHC satisfiability checking based on unfold/fold transformations. Our two main contributions are the following: (1) we prove in a precise mathematical sense that the application of the unfold/fold transformation rules cannot worsen the effectiveness of CHC solvers, and actually these rules are able to enlarge the set of satisfiability problems that can be solved by a given class of CHC solvers; and (2) we provide a specific strategy, called *Predicate Pairing*, for applying the transformation rules with the objective of improving the ability of CHC solvers to prove satisfiability.

The basic idea behind the first contribution is as follows. Similarly to what is introduced in a paper by Bjørner et al. (Bjørner et al. 2015), we consider the notion of the $\mathcal{A}$-*definable* model, which is a model definable in a class $\mathcal{A}$ of first order formulas. Typically, CHC solvers (and, in particular, the solvers based on *predicate abstraction*) look for models in specific classes, such as *linear* (*integer or real*) *arithmetic* formulas, or *quantifier-free array* formulas. While satisfiability is undecidable and not semidecidable, the existence of an $\mathcal{A}$-definable model is semidecidable, as long as the validity problem for the formulas in $\mathcal{A}$ is decidable, and hence solvers that find an $\mathcal{A}$-definable model whenever it exists, can indeed be constructed. We prove that, under very general conditions on $\mathcal{A}$, the unfold/fold rules preserve the existence of an $\mathcal{A}$-definable model, that is, if the original clauses have an $\mathcal{A}$-definable model, then also the transformed clauses have an $\mathcal{A}$-definable model. The converse does not hold: there are cases where the original clauses have

no $\mathcal{A}$-definable model, while the transformed clauses have an $\mathcal{A}$-definable model. In this sense the application of the unfold/fold rules may improve the effectiveness of a CHC solver that works by searching for $\mathcal{A}$-definable models, because the solver may be able to find an $\mathcal{A}$-definable model after the transformation in cases where there was no such a model before the transformation.

We also provide less general conditions under which the transformed clauses have an $\mathcal{A}$-definable model *if and only if* the original ones have an $\mathcal{A}$-definable model. These conditions prevent the introduction of new predicates that have recursive definitions in terms of the old predicates occurring in the original clauses. Thus, the source of the improvement of the effectiveness of the CHC solver due to the unfold/fold transformations is the introduction of one or more new predicates and the derivation of new (mutually) recursive definitions for these predicates.

The second contribution of our paper is related to the fact that, due to the already mentioned undecidability limitations, there is no universal algorithm that, starting from a set of clauses, applies the unfold/fold rules and derives a set of clauses such that, if it is satisfiable, then it has an $\mathcal{A}$-definable model, for some theory $\mathcal{A}$ whose validity problem is decidable. Therefore, it should not be unexpected that the Predicate Pairing strategy we propose for guiding the use of the unfold/fold transformation rules is based on heuristics. We show that this strategy is capable, in many significant cases, of transforming sets of clauses into new, equisatisfiable sets of clauses, whose satisfiability problem can be solved by constructing $\mathcal{A}$-definable models, while the original sets have no $\mathcal{A}$-definable models. Predicate Pairing introduces a new predicate defined by the conjunction of two predicates together with a conjunction of constraints. We will explain through examples why an $\mathcal{A}$-definable model may exist for a conjunction of predicates, even if it does not exist for the atomic conjuncts in isolation. (Obviously, by a repeated application of Predicate Pairing, we may introduce new predicates corresponding to the conjunction of more than two old predicates.) Thus, Predicate Pairing can be viewed as an extension to constrained Horn clauses of techniques for transforming logic programs, such as the *tupling* unfold/fold strategy (Pettorossi and Proietti 1994) and *conjunctive partial deduction* (De Schreye et al. 1999).

We will show that constraint-based reasoning is essential for guiding the introduction of the suitable pairs of predicates during the transformation process. Moreover, we will show that Predicate Pairing works well for solving many satisfiability problems that arise from the field of imperative program verification. In particular, Predicate Pairing is a crucial technique for verifying *relational program properties* (Barthe et al. 2011), that is, properties relating two programs (such as program equivalence) or two executions of the same program (such as non-interference).

The paper is structured as follows. In Section 2 we recall the basic notions concerning constrained Horn clauses and we define the notion of an $\mathcal{A}$-definable model. In Section 3 we prove our results concerning the preservation of $\mathcal{A}$-definable models when using the unfold/fold transformation rules. In Section 4 we present the Predicate Pairing strategy and in Section 5 we show some examples of its application for verifying relational program properties. In Section 6 we report the results obtained by our implementation of that strategy by using the VeriMAP transformation sys-

tem (De Angelis et al. 2014b). Finally, in Section 7 we discuss the related work in the fields of program transformation and verification.

2 Constrained Horn Clauses

In this section we recall the basic definitions concerning constrained Horn clauses and their satisfiability, and we introduce the notion of an $\mathcal{A}$-*definable model*.

Let $\mathcal{L}$ be a first order language with equality and $Pred_u \subseteq \mathcal{L}$ be a set of predicate symbols, called the *user-defined* predicate symbols. Let $\mathcal{C}$ be a set of formulas of $\mathcal{L}\backslash Pred_u$, called the set of *constraints*. We assume that: (i) *true*, *false*, and equalities between terms belong to $\mathcal{C}$, and (ii) $\mathcal{C}$ is closed under conjunction.

An *atom* is an atomic formula of the form $p(X_1, \ldots, X_m)$, where p is a predicate symbol in $Pred_u$ and $X_1, \ldots, X_m$ are distinct variables. Let $Atom$ be the set of all atoms. A *definite constrained clause* is an implication of the form $c \wedge G \rightarrow H$ whose premise (or *body*) is the conjunction of a constraint c and a (possibly empty) conjunction G of $n\,(\geq 0)$ atoms $A_1 \wedge \ldots \wedge A_n$, and whose conclusion (or *head*) H is an atom. A *constrained goal* (or simply, a *goal*) is an implication of the form $c \wedge G \rightarrow \mathit{false}$, where c is a constraint and G is a conjunction of atoms. A *constrained Horn clause* (CHC) (or simply, a *clause*) is either a definite constrained clause or a constrained goal. A set of constrained Horn clauses is said to be a *CHC set*. A constrained Horn clause Cl (or a set P of clauses) is said to be '*over* $\mathcal{C}$' in case we want to stress that the constraints occurring in clause Cl (or in the set P of clauses) are taken from the set $\mathcal{C}$ of constraints. A clause $c \wedge G \rightarrow H$ is said to be *linear* if G consists of at most one atom, and *nonlinear* otherwise.

We will often use the logic programming syntax and we write $H \leftarrow c, A_1, \ldots, A_n$, instead of $c \wedge A_1 \wedge \ldots \wedge A_n \rightarrow H$. We will also feel free to write non-variable terms as arguments of atoms. Thus, the clause $p(\ldots, t, \ldots) \leftarrow c, G$ should be viewed as a shorthand for $p(\ldots, X, \ldots) \leftarrow X = t, c, G$, where X is a variable not occurring elsewhere in the clause, and likewise, $H \leftarrow c, G_1, p(\ldots, t, \ldots), G_2$ should be viewed as a shorthand for $H \leftarrow X = t, c, G_1, p(\ldots, X, \ldots), G_2$.

Given a formula $\varphi \in \mathcal{L}$, we denote by $\exists(\varphi)$ its *existential closure* and by $\forall(\varphi)$ its *universal closure*. By $vars(\varphi)$ and $Fvars(\varphi)$ we denote the set of variables and the set of the free variables, respectively, occurring in φ.

For the notions of an *interpretation* and a *model* of a first order formula we will use the standard notions and notations (Mendelson 1997). We fix a *canonical interpretation* $\mathbb{D}$ of the symbols in $\mathcal{L}\backslash Pred_u$. A $\mathbb{D}$-*interpretation* is an interpretation of $\mathcal{L}$ that for all symbols occurring in $\mathcal{L}\backslash Pred_u$, agrees with $\mathbb{D}$. If U is the universe of $\mathbb{D}$, then a $\mathbb{D}$-interpretation $\mathbb{I}$ can be identified with the set of atoms:

$\{p(a_1, \ldots, a_m) \mid (a_1, \ldots, a_m) \in U^m$ and $p^{\mathbb{I}}(a_1, \ldots, a_m)$ holds in $\mathbb{I}\}$

where $p^{\mathbb{I}}$ denotes the m-ary relation which is the interpretation of p in $\mathbb{I}$. Given any set F of formulas, a $\mathbb{D}$-interpretation $\mathbb{M}$ is a $\mathbb{D}$-*model* of F, written $\mathbb{M} \models F$, if, for all formulas $\varphi \in F$, $\mathbb{M} \models \varphi$ holds. F is $\mathbb{D}$-*satisfiable* if it has a $\mathbb{D}$-model. We will feel free to say *satisfiable*, instead of $\mathbb{D}$-*satisfiable*, when the interpretation $\mathbb{D}$ is clear from the context. We write $\mathbb{D} \models F$ if, for every $\mathbb{D}$-interpretation $\mathbb{M}$, $\mathbb{M} \models F$ holds.

A set P of definite constrained clauses is $\mathbb{D}$-satisfiable and has a *least* (with respect to set inclusion) $\mathbb{D}$-model, denoted $lm(P)$ (Jaffar and Maher 1994). Thus, if P is any set of constrained Horn clauses and Q is the subset of the constrained goals in P, then P is $\mathbb{D}$-satisfiable if and only if $lm(P \setminus Q) \models Q$.

Many CHC solvers based on *predicate abstraction* (Bjørner et al. 2015) try to check the $\mathbb{D}$-satisfiability of a set of constrained Horn clauses by looking for the existence of $\mathbb{D}$-models that are definable by formulas belonging to a given set $\mathcal{A}$, which is a subset of the set $\mathcal{C}$ of constraints. This restriction when looking for models may significantly ease the satisfiability test, as shown by the following example.

Example 1
Let us assume that $\mathcal{C}$ is the set of *linear integer arithmetic* (LIA) constraints, that is, equalities ($=$) and inequalities ($>$) between linear polynomials with integer coefficients and integer-valued variables, closed with respect to conjunction and disjunction. We also use the symbols $\geq$, $\leq$, $<$, and $\neq$ with the usual definitions in terms of $=$ and $>$. Let $\mathbb{Z}$ denote the usual interpretation of integer arithmetic.

Now, let us consider clauses 1–3 listed in the Introduction. The satisfiability of these clauses can be proved by looking for models that are definable by constraints φ in the subset of LIA, which we call 2VAR, defined by the following grammar:

$\varphi ::= \mathit{true} \mid \mathit{false} \mid X > 0 \mid X = 0 \mid X > Y \mid X = Y \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$

where X and Y are variables. Thus, 2VAR is the set of linear integer constraints constructed from arithmetic comparisons between at most two variables, and it is a subset of the *Octagons* domain often considered in the field of abstract interpretation (Miné 2006). A 2VAR-definable model of clauses 1–3 is given by interpreting the predicate $su(M, R, Sum)$ as the set of triples satisfying the following constraint in 2VAR:

$(Sum \geq M \wedge Sum \geq R) \ \vee \ R < 0$,

In order to show that the above interpretation indeed defines a $\mathbb{Z}$-model, we replace the instances of $su(M, R, Sum)$ by the corresponding instances of the formula $(Sum \geq M \wedge Sum \geq R) \vee R < 0$ in clauses 1–3, and we check that the resulting implications hold in $\mathbb{Z}$. □

This Example 1 motivates the introduction of the notion of an $\mathcal{A}$-definable model, which is a generalization of the one presented in the literature (Bjørner et al. 2015), where $\mathcal{A}$ coincides with $\mathcal{C}$.

Definition 1
Let $\mathcal{A} \subseteq \mathcal{C}$ be a set of formulas of $\mathcal{L} \setminus Pred_u$ such that: (i) *true*, *false*, and equalities between terms belong to $\mathcal{A}$, and (ii) $\mathcal{A}$ is closed under conjunction. Let $\mathbb{D}$ be the canonical interpretation of the symbols in $\mathcal{L} \setminus Pred_u$. We denote by $\mathcal{A}^{\exists\vee}$ the set of formulas $\{\exists X_1 \ldots \exists X_m (\varphi_1 \vee \ldots \vee \varphi_n) \mid m \geq 0, n > 0,$ and for $i = 1, \ldots, n, \ \varphi_i \in \mathcal{A}\}$. A *symbolic interpretation* is a function $\Sigma\colon Atom \cup \{\mathit{false}\} \longrightarrow \mathcal{A}^{\exists\vee}$ such that $\Sigma(\mathit{false}) = \mathit{false}$ and, for every $A \in Atom$, (i) $Fvars(\Sigma(A)) \subseteq Fvars(A)$, and (ii) for every renaming substitution ρ for A (Lloyd 1987), $\Sigma(A\rho) = \Sigma(A)\rho$. We extend Σ to conjunctions of atoms by stating that $\Sigma(A_1 \wedge \ldots \wedge A_n) = \Sigma(A_1) \wedge \ldots \wedge \Sigma(A_n)$. Given a set P of constrained Horn clauses over $\mathcal{C}$, a symbolic interpretation Σ is an

$\mathcal{A}$-definable model of P, written $\Sigma \models P$, if for every clause $H \leftarrow c, A_1, \ldots, A_n$ in P, $\mathbb{D} \models \forall(c \wedge \Sigma(A_1 \wedge \ldots \wedge A_n) \rightarrow \Sigma(H))$ holds.

Note that the symbolic interpretation Σ of an atom is independent of the variable names occurring in that atom, and hence, for each predicate symbol p, the formula $\Sigma(p(X_1, \ldots, X_m))$ is unique up to variable renaming. Note also that the definition of a symbolic interpretation is essentially equivalent to that given by Kafle and Gallagher, who define an interpretation as a set of *constrained facts* of the form $p(X_1, \ldots, X_m) \leftarrow c$, where c is a constraint in $\mathcal{A}$ (Kafle and Gallagher 2017b). Indeed, the set of constraints in the bodies of the constrained facts with the same head predicate can be represented as a disjunction of those constraints, and the variables occurring in the body of a constrained fact and not in its head are implicitly existentially quantified.

Clearly, if P has an $\mathcal{A}$-definable model, then P is $\mathbb{D}$-satisfiable. In general the converse does not hold, as shown by the following example.

Example 2
Let us continue Example 1, where the sets of constraints $\mathcal{C}$ and $\mathcal{A}$ are LIA and 2VAR, respectively. Let us consider the program in Figure 2, which computes the square of a non-negative integer `n` by summing up `n` times the value of `n`.

```
int n, sqr;
int g(int y, int k) {
  int s = 0;
  while (y > 0) {
    s = s + k;   y--; }
  return s;
  }
void square() {
  sqr = g(n,n);
  }
```

Figure 2. Program `square` computing $\mathtt{sqr} = \mathtt{n}^2$.

Clauses 4–6 below express the following property, which relates program `sum_upto` and program `square`: if the value of `m` is equal to the value of `n` before the execution of the programs `sum_upto` and `square` and they both terminate, then at the end of their execution the value of `sqr` is not smaller than the value of `sum`. Note that, since the programs `sum_upto` and `square` have disjoint sets of variables, the order of their execution is immaterial.

4. $\mathit{false} \leftarrow \mathit{Sum} > \mathit{Sqr},\ M \geq 0,\ M = N,\ N = Y,\ R0 = 0,\ S0 = 0,$
   $\mathit{su}(M, R0, \mathit{Sum}),\ \mathit{sq}(N, Y, S0, \mathit{Sqr})$
5. $\mathit{sq}(K, Y, S0, S) \leftarrow Y \leq 0,\ S = S0$
6. $\mathit{sq}(K, Y, S0, S) \leftarrow Y > 0,\ Y1 = Y - 1,\ S1 = S0 + K,\ \mathit{sq}(K, Y1, S1, S)$

For $Y \geq 0$, the atom $\mathit{sq}(K, Y, S0, S)$ holds iff $S = S0 + (K \times Y)$. Properties like the one between programs `sum_upto` and `square` are called *relational properties* (Barthe et al. 2011). Similarly to the verification conditions for partial correctness properties, the clauses for relational properties, also called verification conditions,

can be automatically generated from the formal specification of those properties and the operational semantics of the programming language (De Angelis et al. 2016). For brevity, we do not give here the details of that generation process, which is inessential for understanding the techniques presented in this paper.

Clauses 2–6 are constrained Horn clauses over LIA and they are $\mathbb{Z}$-satisfiable. Indeed, in the least $\mathbb{Z}$-model of clauses 2, 3, 5, 6, for all integers M, Sum, and Sqr, with $M \geq 0$, if $su(M, 0, Sum)$ and $sq(M, M, 0, Sqr)$ hold, then $Sum \leq Sqr$ holds. However, clauses 2–6 do not admit a 2VAR-definable model. Indeed, no constraint of the form $Sum \leq X$, for any variable X, is a consequence of $su(M, 0, Sum)$, and hence we cannot infer $Sum \leq Sqr$, independently of the constraints that are consequences of $sq(M, M, 0, Sqr)$. Actually, it is not difficult to see that a similar limitation holds even if we look for a LIA-definable model, rather that a 2VAR-definable model. Indeed, in order to infer the constraint $Sum \leq Sqr$ one should discover quadratic relations, such as $Sum = M \times (M-1)/2$ and $Sqr = M \times M$, starting from $su(M, 0, Sum)$ and $sq(M, M, 0, Sqr)$, respectively, and these relations cannot be expressed by linear arithmetic constraints. □

3 Transformation Rules and Preservation of $\mathcal{A}$-definable Models

Let $\mathcal{C}$ be a set of constraints and $\mathcal{A} \subseteq \mathcal{C}$. A *transformation sequence over* $\mathcal{C}$ is a sequence of CHC sets $P_0, P_1, \ldots, P_n$ over $\mathcal{C}$, where, for $i = 0, \ldots, n-1$, P_{i+1} is derived from P_i by applying one of the following rules R1–R4.

Let $Defs_i$ denote the set of all the clauses, called *definitions*, introduced by rule R1 during the construction of the transformation sequence $P_0, P_1, \ldots, P_i$. Thus, $Defs_0 = \emptyset$.

(R1) *Definition.* We introduce a clause D: $newp(X_1, \ldots, X_k) \leftarrow c, G$, where: (i) $newp$ is a predicate symbol in $Pred_u$ not occurring in the sequence $P_0, P_1, \ldots, P_i$, (ii) $c \in \mathcal{A}$, (iii) G is a non-empty conjunction of atoms whose predicate symbols occur in P_0, and (iv) $X_1, \ldots, X_k$ are distinct variables occurring free in (c, G). Then, we derive the new set $P_{i+1} = P_i \cup \{D\}$ and $Defs_{i+1} = Defs_i \cup \{D\}$.

(R2) *Unfolding.* Let C: $H \leftarrow c, G_1, p(X_1, \ldots, X_k), G_2$ be a clause in P_i. Let
$\{p(X_1, \ldots, X_k) \leftarrow c_j, B_j \mid j = 1, \ldots, m\}$
be the (possibly empty) set of clauses in P_i whose head predicate is p. Without loss of generality, we assume that, for $j = 1, \ldots, m$, $vars(c_j, B_j) \cap vars(C) \subseteq \{X_1, \ldots, X_k\}$. By *unfolding the atom* $p(X_1, \ldots, X_k)$ *in* C *using* P_i we derive the new set $P_{i+1} = (P_i \setminus \{C\}) \cup \{H \leftarrow c, c_j, G_1, B_j, G_2 \mid j = 1, \ldots, m\}$.

(R3) *Folding.* Let C: $H \leftarrow c, G_1, Q, G_2$ be a clause in P_i, where Q is a non-empty conjunction of atoms, and let D: $K \leftarrow d, B$ be (a variant of) a clause in $Defs_i$ with $vars(C) \cap vars(D) = \emptyset$. Suppose that there exist a substitution ϑ and a constraint e such that: (i) $Q = B\vartheta$, (ii) $\mathbb{D} \models \forall(c \leftrightarrow (e \wedge d\vartheta))$, and (iii) for every variable $X \in vars(d, B) \setminus vars(K)$, the following conditions hold: (iii.1) $X\vartheta$ is a variable not occurring in $\{H, c, G_1, G_2\}$, and (iii.2) $X\vartheta$ does not occur in the term $Y\vartheta$, for any variable Y occurring in (d, B) and different from X. By *folding* C *using the*

definition D, we derive clause E: $H \leftarrow e, G_1, K\vartheta, G_2$. In this case we also say that E is derived *by folding* Q *in* C. We derive the new set $P_{i+1} = (P_i \setminus \{C\}) \cup \{E\}$.

(R4) *Constraint Replacement.* Let us consider a subset of P_i of the form $\{(H \leftarrow c_1, G), \ldots, (H \leftarrow c_k, G)\}$. Suppose that, for some constraints $d_1, \ldots, d_m$,

$\mathbb{D} \models \forall\, (\exists\, Y_1 \ldots \exists\, Y_r\ (c_1 \vee \ldots \vee c_k) \leftrightarrow \exists Z_1 \ldots \exists Z_s\ (d_1 \vee \ldots \vee d_m))$

where $\{Y_1, \ldots, Y_r\} = Fvars(c_1 \vee \ldots \vee c_k) \setminus vars(\{H, G\})$ and $\{Z_1, \ldots, Z_s\} = Fvars(d_1 \vee \ldots \vee d_m) \setminus vars(\{H, G\})$. Then, we derive the new set $P_{i+1} = (P_i \setminus \{(H \leftarrow c_1, G), \ldots, (H \leftarrow c_k, G)\}) \cup \{(H \leftarrow d_1, G), \ldots, (H \leftarrow d_m, G)\}$.

Note that rule R4 enables the deletion of a clause with an inconsistent constraint in its body. Indeed, if c_1 is unsatisfiable, then $\mathbb{D} \models \forall\, (c_1 \leftrightarrow d_1 \vee \ldots \vee d_m)$ with $m = 0$.

The following result (Etalle and Gabbrielli 1996) shows that the transformation rules R1–R4 derive sets of clauses that are equivalent with respect to the least $\mathbb{D}$-model.

Theorem 1 (*Equivalence with respect to the Least* $\mathbb{D}$*-Model*)
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence where, for $i = 0, \ldots, n$, P_i is a set of definite clauses. Let us assume that every definition in $Defs_n$ is unfolded during the construction of this sequence (that is, for every definition $D \in Defs_n$, there exists i, with $0 \leq i \leq n-1$, such that P_{i+1} is derived from P_i by unfolding D). Then, for every predicate p and $(a_1, \ldots, a_m) \in U^m$,

$p(a_1, \ldots, a_m) \in lm(P_0 \cup Defs_n)$ if and only if $p(a_1, \ldots, a_m) \in lm(P_n)$.

From Theorem 1 it follows that, as we now show, the transformation rules R1–R4 derive sets of clauses that are equivalent with respect to $\mathbb{D}$-satisfiability.

Theorem 2 (*Equivalence with respect to* $\mathbb{D}$*-Satisfiability*)
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence such that every definition in $Defs_n$ is unfolded during the construction of this sequence. Then, P_0 is $\mathbb{D}$-satisfiable if and only if P_n is $\mathbb{D}$-satisfiable.

Proof
First we observe that P_0 is $\mathbb{D}$-satisfiable iff $P_0 \cup Defs_n$ is $\mathbb{D}$-satisfiable. Indeed: (i) if M is a $\mathbb{D}$-model of P_0, then the $\mathbb{D}$-interpretation $M \cup \{newp(a_1, \ldots, a_k) \mid newp$ is a head predicate in $Defs_n$ and $(a_1, \ldots, a_k) \in U^k\}$ is a $\mathbb{D}$-model of $P_0 \cup Defs_n$, and (ii) if M is a $\mathbb{D}$-model of $P_0 \cup Defs_n$, then by the definition of $\mathbb{D}$-model, M is a $\mathbb{D}$-model of P_0.

Now let us consider a new sequence $P'_0, P'_1, \ldots, P'_n$ obtained from the transformation sequence $P_0, P_1, \ldots, P_n$ by replacing each occurrence of *false* in the head of a clause by a new predicate symbol f. The sequence $P'_0, P'_1, \ldots, P'_n$ satisfies the hypothesis of Theorem 1, and hence $f \in lm(P'_0 \cup Defs_n)$ iff $f \in lm(P'_n)$.

We have that: $P_0 \cup Defs_n$ is $\mathbb{D}$-satisfiable iff $P'_0 \cup Defs_n \cup \{\neg f\}$ is $\mathbb{D}$-satisfiable iff $f \notin lm(P'_0 \cup Defs_n)$ iff {by Theorem 1} $f \notin lm(P'_n)$ iff $P'_n \cup \{\neg f\}$ is $\mathbb{D}$-satisfiable iff P_n is $\mathbb{D}$-satisfiable. □

Theorem 2 is *not* sufficient to ensure that a transformation sequence preserves the existence of an $\mathcal{A}$-definable model. Indeed, as shown by Example 2 in Section 2,

for some set $\mathcal{A}$ of constraints, $\mathbb{D}$-satisfiability does not imply the existence of an $\mathcal{A}$-definable model.

Now, in order to study the preservation of $\mathcal{A}$-models during the construction of a transformation sequence over $\mathcal{C}$, for $\mathcal{A} \subseteq \mathcal{C}$, we introduce the notions of *$\mathcal{A}$-soundness* and *$\mathcal{A}$-completeness*.

Definition 2 (*$\mathcal{A}$-Soundness, $\mathcal{A}$-Completeness*)
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence. (i) If P_0 has an $\mathcal{A}$-definable model implies that P_n has an $\mathcal{A}$-definable model, we say that the sequence is *$\mathcal{A}$-sound*. (ii) If P_n has an $\mathcal{A}$-definable model implies that P_0 has an $\mathcal{A}$-definable model, we say that the sequence is *$\mathcal{A}$-complete*.

In order to prove the $\mathcal{A}$-soundness of a transformation sequence (see Theorem 4 below) we need the following definition and theorem.

Definition 3
Let $\mathcal{S}$ be a CHC set. A symbolic interpretation Σ is said to be *tight on* $\mathcal{S}$ if for all clauses $A \leftarrow c, G$ in $\mathcal{S}$, $\mathbb{D} \models \forall(\Sigma(A) \leftrightarrow \exists X_1 \ldots \exists X_k(c \wedge \Sigma(G)))$, where $\{X_1, \ldots, X_k\} = \mathit{Fvars}(c \wedge \Sigma(G)) \backslash \mathit{Fvars}(\Sigma(A))$.

For instance, given the singleton set of clauses $\mathcal{S} = \{p(X) \leftarrow q(X)\}$, the symbolic interpretation Σ_1 that maps both $p(X)$ and $q(X)$ to $X = 0$ is tight on $\mathcal{S}$, while the symbolic interpretation Σ_2 that maps $p(X)$ to *true* and $q(X)$ to $X = 0$ is not tight on $\mathcal{S}$. Both Σ_1 and Σ_2 are models.

Theorem 3
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence. For $i = 0, \ldots, n-1$, if P_i has an $\mathcal{A}$-definable model that is tight on Defs_i, then P_{i+1} has an $\mathcal{A}$-definable model that is tight on Defs_{i+1}.

Proof See Appendix. □

The hypothesis that P_i has an $\mathcal{A}$-definable model that is tight on Defs_i is needed to guarantee that the folding rule replaces a conjunction consisting of constraints and atoms by a single atom which is *equivalent* in the given model.

From Theorem 3 and the fact that, if P_0 has an $\mathcal{A}$-definable model, then P_0 has an $\mathcal{A}$-definable model that is tight on Defs_0, which is the empty set, we get the following result.

Theorem 4 (*$\mathcal{A}$-Soundness*)
Every transformation sequence is $\mathcal{A}$-sound.

Now we prove that, if some suitable hypotheses hold, a transformation sequence is also $\mathcal{A}$-complete. First, we need the following two definitions.

Definition 4
An application of the unfolding rule R2 to a clause C: $H \leftarrow c, G_1, p(X_1, \ldots, X_k), G_2$ in P_i is said to be a *self-unfolding* if the predicate of H is p.

Definition 5
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence. An application of the folding rule R3 to a clause C in P_i using a definition D in Defs_i is said to be a *reversible folding* if D belongs to P_i and is different from C.

Theorem 5 ($\mathcal{A}$*-Completeness*)
Let $P_0, P_1, \ldots, P_n$ be a transformation sequence over a set $\mathcal{C}$ of constraints. Let $\mathcal{A}$ be equal to $\mathcal{C}$. Suppose that: (i) no application of the unfolding rule is a self-unfolding, and (ii) every application of the folding rule is a reversible folding. Then, $P_0, P_1, \ldots, P_n$ is $\mathcal{A}$-complete.

Proof See Appendix. □

In Section 4 we will present the Predicate Pairing transformation strategy and we will show that it generates transformation sequences that are $\mathcal{A}$-sound, but not necessarily $\mathcal{A}$-complete (see Theorem 6).

Normally, $\mathcal{A}$-soundness is a desirable property of a transformation sequence $P_0, P_1, \ldots, P_n$. Indeed, suppose we have a CHC solver, call it *SOLVE*, that finds an $\mathcal{A}$-definable model of a set of clauses whenever it exists. As already mentioned, such an ideal solver exists, as long as the validity problem for the formulas in $\mathcal{A}$ is decidable. Then, $\mathcal{A}$-soundness guarantees that if the satisfiability of P_0 can be proved by using *SOLVE*, then also the satisfiability of P_n can be proved by using *SOLVE*. In other terms, the effectiveness of the solver is not worsened by the transformation.

In contrast, $\mathcal{A}$-completeness might not always be a desirable property. Indeed, in many cases we may want to transform clauses for which *SOLVE* cannot find an $\mathcal{A}$-definable model, because such a model does not exist, and derive equisatisfiable clauses with an $\mathcal{A}$-definable model which can be constructed by using *SOLVE*.

In practice, the existing solvers do not guarantee that they find an $\mathcal{A}$-definable model of a set of clauses whenever it exists. Thus, the theoretical properties of $\mathcal{A}$-soundness and $\mathcal{A}$-completeness might not hold in some cases. We will show through the experiments reported in Section 6, that these unfortunate cases are rare.

We conclude this section by showing that there are $\mathbb{D}$-safisfiable CHC sets that have no $\mathcal{A}$-definable models, and yet can be transformed, by applying rules R1–R4, into CHC sets that have $\mathcal{A}$-definable models.

Example 3
Let us continue Example 2, where $\mathbb{D}$ is $\mathbb{Z}$, $\mathcal{C}$ is LIA, and $\mathcal{A}$ is 2VAR. Let P_0 be the set consisting of clauses 2–6.

Starting from P_0 we construct a transformation sequence P_0, P_1, P_2, P_3, P_4, as we now indicate. First, by applying the definition rule, we introduce the following new predicate:

7. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M = Y,\ su(M, R0, Sum),\ sq(N, Y, S0, Sqr)$

We derive the clause set $P_1 = P_0 \cup \{7\}$ and $Defs_1 = \{7\}$. Now, by unfolding the atoms su and sq of clause 7, and then performing some more unfoldings of the derived atoms, we get:

8. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M = Y,\ M \leq 0,\ Sum = R0,\ Sqr = S0$
9. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M = Y,\ M \leq 0,\ Sum = R0,\ Y > 0,$
   $Y1 = Y - 1,\ S1 = S0 + N,\ sq(N, Y1, S1, Sqr)$
10. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M = Y,\ M > 0,\ M1 = M - 1,\ R1 = R0 + M,$
    $Y \leq 0,\ Sqr = S0,\ su(M1, R1, Sum)$

11. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M = Y,\ M > 0,\ M1 = M - 1,\ R1 = R0 + M,$
$Y > 0,\ Y1 = Y - 1,\ S1 = S0 + N,\ su(M1, R1, Sum),\ sq(N, Y1, S1, Sqr)$

We get $P_2 = P_0 \cup \{8, 9, 10, 11\}$ and $Defs_2 = \{7\}$. (Here and in the rest of the example, for reasons of conciseness, we feel free to avoid to list some intermediate CHC sets in the transformation sequence.) By the constraint replacement rule R4 we can remove clauses 9 and 10, whose bodies have unsatisfiable constraints, and replace clauses 8 and 11 by:

12. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M \leq 0,\ Sum = R0,\ Sqr = S0$
13. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M > 0,\ M1 = M - 1,\ R1 = R0 + M,$
$S1 = S0 + N,\ M1 = Y1,\ su(M1, R1, Sum),\ sq(N, Y1, S1, Sqr)$

We get $P_3 = P_0 \cup \{12, 13\}$ and $Defs_3 = \{7\}$. Then, by the folding rule R3, we fold clause 4 (in P_0) using clause 7 and we derive the following clause:

14. $false \leftarrow Sum > Sqr, M \geq 0, M = N, R0 = 0, S0 = 0, su\_sq(M, R0, Sum, N, S0, Sqr)$

Finally, we fold clause 13 using clause 7 and we derive the following clause:

15. $su\_sq(M, R0, Sum, N, S0, Sqr) \leftarrow M > 0,\ M1 = M - 1,\ R1 = R0 + M,$
$S1 = S0 + N,\ su\_sq(M1, R1, Sum, N, S1, Sqr)$

We get the final set of clauses $P_4 = (P_0 \setminus \{4\}) \cup \{12, 14, 15\}$. Now, it is easy to check that the symbolic interpretation that maps the atom $su\_sq(M, R0, Sum, N, S0, Sqr)$ to the 2VAR constraint $Sum \leq Sqr \vee R0 > S0 \vee M > N$, and the su and sq atoms to $true$, is a 2VAR-definable model of P_4. This check can be done by replacing the atoms in P_4 by the corresponding symbolic interpretations, and then verifying the validity of the formulas obtained in that way by using an SMT solver for linear arithmetic, such as the popular Z3 solver (de Moura and Bjørner 2008).

Let us now make some remarks on the derivation above.

(1) The applications of the transformation rules satisfy the hypothesis of Theorem 2 and hence P_0 is $\mathbb{Z}$-satisfiable if and only if P_4 is $\mathbb{Z}$-satisfiable (indeed, they are both $\mathbb{Z}$-satisfiable).

(2) The fact that P_4 has a 2VAR-definable model, while P_0 has no such model, is due to the fact that the applications of the folding rule R3 are not reversible foldings, and hence the transformation sequence $P_0, \ldots, P_4$ does not satisfy the hypothesis of Theorem 5. Indeed, clause 7 occurs in $Defs_3$, but not in P_3. More in general, the derivation of a CHC set with an $\mathcal{A}$-definable model from a CHC set without an $\mathcal{A}$-definable model is due to the introduction of new predicates and also to the derivation (via non-reversible foldings) of clauses that constitute recursive definitions of these new predicates.

(3) Finally, note that at every step during the transformation from P_0 to P_4 we have handled linear constraints only. However, the introduction of the new predicate $su\_sq(M, R0, Sum, N, S0, Sqr)$, defined in terms of the conjunction of $su(M, R0, Sum)$ and $sq(N, Y, S0, Sqr)$, allows us to discover linear relations between Sum and Sqr without having to deal with nonlinear constraints. □

4 Predicate Pairing

In Section 3 we have seen that by the applying unfold/fold rules, in some cases one may transform a given $\mathbb{D}$-satisfiable set of clauses which does *not* admit an $\mathcal{A}$-definable model, into an equisatisfiable set of clauses which admits an $\mathcal{A}$-definable model. Then, the $\mathbb{D}$-satisfiability of the set of clauses can be proved by a CHC solver that constructs an $\mathcal{A}$-definable model. Example 3 of Section 3 suggests that the crucial step in that transformation is a predicate pairing step, that is, the introduction of a new predicate, say t, whose defining clause has in its body the conjunction of two atoms, one with predicate, say q, and the other with predicate, say r, whose definitions are provided by the original set of clauses. Indeed, the predicate pairing allows us to derive suitable relations between arguments of the conjunction of q and r, which cannot be expressed by using constraints on the arguments of q and r separately.

In general, in order to do the transformation and perform some required folding steps, it may be necessary to introduce, by predicate pairing, more than one definition. The introduction of these new definitions is a major issue in the case where the predicates to be paired should be chosen from the various predicates occurring in the conjunction of several atoms in the body of a clause. In particular this issue arises when predicates are defined by *nonlinear* clauses, and hence their repeated unfolding may generate unbounded conjunctions of atoms.

In this section we will present a strategy, called *Predicate Pairing* (or PP, for short), for making the choice of the predicates to be paired. This strategy, which is realized by Algorithm 1, takes as input a set of clauses and derives a new, equisatisfiable set of clauses, which by Theorem 4 is guaranteed to admit an $\mathcal{A}$-definable model, whenever the original clauses had one. Actually, as we will show, in many interesting cases the Predicate Pairing strategy constructs new sets of clauses for which a CHC solver is able to construct one such model, while the same solver is unable to do so for the original set of clauses. We present the Predicate Pairing strategy with the help of an example. Suppose that we are given the following specification of (a variant of) the Ackermann function:

S1. $ackermann(m,n) = n+1$ *if* $m \leq 0$

S2. $ackermann(m,n) = ackermann(m-1, 1)$ *if* $m > 0 \wedge n = 0$

S3. $ackermann(m,n) = ackermann(m-1, ackermann(m, n-1))$ *if* $m > 0 \wedge n > 0$

and the following two programs each of which implements that specification:

```
int ackermann1(int m, int n) {
    if (m =< 0) { return n+1; }
    else if (m > 0 && n = 0) { ackermann1(m-1,1); }
    else if (m > 0 && n > 0) { ackermann1(m-1,ackermann1(m,n-1)); }
  }
int ackermann2(int m, int n) {
  while (m > 0) {
    if (n == 0) { m = m-1;   n = 1; }
    else { n = ackermann2(m,n-1);   m = m-1; }
  }
  return n+1;
}
```

Algorithm 1: The *Predicate Pairing* (PP) strategy.

Input: (i) a clause C_{init} of the form: $false \leftarrow c_{init}, q(X), r(Y)$, and
(ii) two disjoint sets Q and R of clauses such that: q occurs in Q, r occurs in R, and Q and R have no predicates in common.
Output: a set *TransfCls* of clauses such that there is no occurrence of an atom with predicate in Q and an atom with predicate in R in the same clause body.

Notation:
For all constraints d, for all atoms A and B,
let $Eq(d,A,B)$ be $\{X{=}Y \mid X{\in}\, vars(A),\ Y{\in}\, vars(B),\ \mathbb{D} \models \forall(d \rightarrow (X{=}Y))\}$.

$InCls := \{C_{init}\}$; $Defs := \emptyset$; $TransfCls := Q \cup R$;
while in $InCls$ there is a clause C of the form: $L \leftarrow c, A_Q, B_R$, where:
(i) A_Q occurs in Q, and (ii) B_R occurs in R **do**

- UNFOLDING: Unfold once the atom A_Q and once the atom B_R in clause C using $Q \cup R$, thereby deriving the set *UnfoldedCls* of clauses;
- DEFINITION & FOLDING: $FoldedCls := UnfoldedCls$;
  while in *FoldedCls* there is clause E of the form: $H \leftarrow d, A, B, G$, where:
  (i) A and B are atoms whose predicates occur in Q and R, respectively, and
  (ii) G is a conjunction of atoms, such that:
  for all atoms M and N in (A,B,G), whose predicates occur in Q and R, respectively, we have that $|Eq(d,A,B)| \geq |Eq(d,M,N)|$ **do**
  - **if** in *Defs* there is a clause D' of the form: $H' \leftarrow d', A', B'$ such that, for some substitution ϑ, we can fold (A, B) in clause E using D',
    then $FoldedCls := (FoldedCls \backslash \{E\}) \cup \{H \leftarrow d, H'\vartheta, G\}$;
    else
    let D be the clause $newp(Z) \leftarrow e, A, B$, where:
    (i) *newp* is a predicate symbol not occurring elsewhere,
    (ii) $Z = vars(A) \cup vars(B)$, and
    (iii) e is the conjunction of the equalities in $Eq(d,A,B)$;
    $FoldedCls := (FoldedCls \backslash \{E\}) \cup \{H \leftarrow d, newp(Z), G\}$;
    $Defs := Defs \cup \{D\}$; $InCls := InCls \cup \{D\}$;
  $InCls := InCls \backslash \{C\}$; $TransfCls := TransfCls \cup FoldedCls$;

We want to prove the equivalence of these two implementations, in the sense that, for all non-negative integers `m`$\geq$`0` and `n`$\geq$`0`, `ackermann1(m,n)` returns the same integer returned by `ackermann2(m,n)`.

Given the programs `ackermann1(m,n)` and `ackermann2(m,n)`, we first generate the following two sets Q and R of clauses that encode the operational semantics of the programs. These sets of clauses can be derived by specializing the interpreter of the imperative language with respect to the programs (De Angelis et al. 2017).

Q: 1. $ackermann1(M1, N1, A1) \leftarrow ack1(M1, N1, A1)$
2. $ack1(M1, N1, A1) \leftarrow M1 \leq 0, A1 = N1+1$
3. $ack1(M1, N1, A1) \leftarrow M1 > 0, N1 = 0, X1 = M1-1, Y1 = 1, ack1(X1, Y1, A1)$
4. $ack1(M1, N1, A1) \leftarrow M1 > 0, N1 > 0, X1 = M1-1, Y1 = N1-1,$
$ack1(M1, Y1, Z1),\ ack1(X1, Z1, A1)$

R: 5. $ackermann2(M2, N2, A2) \leftarrow A3+1 = A2, ack2(M2, N2, A3)$
6. $ack2(M2, N2, A2) \leftarrow M2 \leq 0, A2 = N2$
7. $ack2(M2, N2, A2) \leftarrow M2 > 0, N2 = 0, M2 = X2+1, Y2 = 1, ack2(X2, Y2, A2)$
8. $ack2(M2, N2, A2) \leftarrow M2 > 0, N2 \neq 0, X2 = M2-1, Y2 = N2-1, Z2 = Z3-1,$
$ack2(M2, Y2, Z2),\ ack2(X2, Z3, A2)$

The equivalence of the functions computed by the programs for `ackermann1` and `ackermann2` is expressed in terms of the predicates defined by clauses 1–8 as follows: for all integers $M1$, $M2$, $N1$, $N2$, we have that if $M1 \geq 0$, $M1 = M2$, $N1 \geq 0$, $N1 = N2$, and $ackermann1(M1, N1, A1)$ and $ackermann2(M2, N2, A2)$ both hold, then $A1 = A2$ holds. Thus, given the clause:

9. $false \leftarrow A1 \neq A2,\ M1 \geq 0,\ M1 = M2,\ N1 \geq 0,\ N1 = N2,$
$ackermann1(M1, N1, A1),\ ackermann2(M2, N2, A2)$

the proof that `ackermann1` and `ackermann2` are equivalent is reduced to the construction of a model for clauses 1–9 (note that the constraint $A1 \neq A2$ in clause 9 states that the values returned by the two programs are different).

Now we have that no CHC solver that constructs LIA-definable models, can prove the satisfiability of clauses 1–9. Indeed, in order to make that proof, the solver should discover that the atoms $ackermann1(M1, N1, A1)$ and $ackermann2(M2, N2, A2)$ imply the two equalities $A1 = ackermann(M1, N1)$ and $A2 = ackermann(M2, N2)$, respectively, where *ackermann* is the function specified by equations $S1$–$S3$, and these equalities cannot be expressed as linear integer constraints.

Thus, in order to allow a CHC solver to construct a LIA-definable model for clauses 1–9, one should avoid reasoning on the two predicates $ackermann1$ and $ackermann2$ in a separate way, and instead, one should reason on the *conjunction* of those predicates. Indeed, in what follows we will derive for that conjunction a new, equisatisfiable set of clauses that has a LIA-definable model. In this new set of clauses we will discover suitable LIA constraints relating the arguments of the predicates $ackermann1(M1, N1, A1)$ and $ackermann2(M2, N2, A2)$. Using these constraints the CHC solver Z3 can show the existence of a LIA-definable model for clauses 1–9, thereby proving the desired equivalence between programs `ackermann1` and `ackermann2`.

This new set of clauses will be derived from clause 9 by applying a sequence of transformation rules according to the Predicate Pairing strategy, as indicated in Algorithm 1. The algorithm takes as input a set of clauses $\{C_{init}\} \cup Q \cup R$, that is, $\{9\} \cup \{1, 2, 3, 4\} \cup \{5, 6, 7, 8\}$ in our case, and produces as output a new set of clauses by applying the unfolding, definition, and folding rules. During the application of that strategy we silently apply the constraint replacement rule to remove clauses which have unsatisfiable constraints in their body.

• *First iteration of the body of the while-loop of Predicate Pairing.*

Since $InCls = \{9\}$, $ackermann1$ occurs in Q, and $ackermann2$ occurs in R, we start off by unfolding $ackermann1(M1, N1, A1)$ and $ackermann2(M2, N2, A2)$ in clause 9. These unfoldings correspond to a symbolic evaluation step of each of the two atoms. We get:

10. $false \leftarrow A1 \neq A2,\ M1 \geq 0,\ M1 = M2,\ N1 \geq 0,\ N1 = N2,\ A2 = A3+1,$
$ack1(M1, N1, A1), ack2(M2, N2, A3)$

Then in order to fold clause 10, we introduce the following definition clause 11 which pairs together the atoms with predicate $ack1$ occurring in Q and predicate $ack2$ occurring in R. In the body of this definition we have the equality constraints $M1 = M2$ and $N1 = N2$ between the arguments of $ack1$ and $ack2$.

11. $new1(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, N1 = N2,$
$ack1(M1, N1, A1), ack2(M2, N2, A2)$

The definition of $new1$ is then used for replacing, by folding, the conjunction of the atoms with predicates $ack1$ and $ack2$ in the body of clause 10. Thus, from clause 10, by folding, we derive:

12. $false \leftarrow A1 \neq A2,\ M1 \geq 0,\ M1 = M2,\ N1 \geq 0,\ N1 = N2,\ A2 = A3+1,$
$new1(M1, N1, A1, M2, N2, A3)$

• *Second iteration of the body of the while-loop of Predicate Pairing.*

Since $InCls = \{11\}$, $ack1$ occurs in Q, and $ack2$ occurs in R, we have to perform a second iteration of the body of the while-loop of the Predicate Pairing.

We unfold the atoms with predicate $ack1$ and $ack2$ in the premise of clause 11, and we get the following three clauses:

13. $new1(M1, N1, A1, M2, N2, N2) \leftarrow M1 \leq 0, M1 = M2, N1 = N2, A1 = N1+1$
14. $new1(M1, N1, A1, M2, N2, A2) \leftarrow M1 > 0, M1 = M2, N1 = 0, N1 = N2,$
$X1 = M1-1, Y1 = 1, X2 = M2-1, Y2 = 1, ack1(X1,Y1,A1), ack2(X2,Y2,A2)$
15. $new1(M1, N1, A1, M2, N2, A2) \leftarrow M1 > 0, M1 = M2, N1 > 0, N1 = N2, N2 \neq 0,$
$Y1 = N1-1, X1 = M1-1, Y2 = N2-1, X2 = M2-1, Z3 = Z2+1,$
$ack1(M1, Y1, Z1), ack1(X1, Z1, A1), ack2(M2, Y2, Z2), ack2(X2, Z3, A2)$

Clause 13 need not be folded because it has no atoms in its body. Clause 14 can be folded using clause 11 (the conditions for folding given in Section 3 are indeed satisfied), and we get:

16. $new1(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 = N2, N1 = 0,$
$X1 = M1-1, Y1 = 1, X2 = M2-1, Y2 = 1, new1(X1, Y1, A1, X2, Y2, A2)$

Clause 15 should be folded, but first we need to choose the atoms to be paired together. According to our goal of proving a relation between the arguments of $ackermann1$ and $ackermann2$, we should pair together an $ack1$ atom with an $ack2$ atom. However, the choice of the atoms to be paired can be made in different ways because in clause 15 there are two $ack1$ atoms and two $ack2$ atoms. The strategy we propose looks at the arguments of the atoms and selects the two atoms which share a maximal number of equality constraints holding between an argument of $ack1$ and an argument of $ack2$.

According to this strategy we have that $ack1(M1, Y1, Z1)$ should be paired with $ack2(M2, Y2, Z2)$ because these two atoms share the two equalities $M1 = M2$

and $Y1 = Y2$ (this last equality follows from $N1 = N2$), while $ack1(M1, Y1, Z1)$ shares no equalities with $ack2(X2, Z3, A2)$. Moreover, $ack1(X1, Z1, A1)$ shares one equality only, namely $X1 = X2$ (this equality follows from $M1 = M2$) with $ack2(X2, Z3, A2)$. Thus, we pair $ack1(M1, Y1, Z1)$ with $ack2(M2, Y2, Z2)$ and then we take clause 11 for folding these two atoms.

In order to fold the other two atoms occurring in clause 15, that is, $ack1(X1,Z1,A1)$ and $ack2(X2, Z3, A2)$, we introduce the following clause:

17. $new2(M1,N1,A1,M2,N2,A2) \leftarrow M1 = M2, ack1(M1,N1,A1), ack2(M2,N2,A2)$

Thus, by folding clause 15 using clauses 11 and 17, we get:

18. $new1(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 = N2, N1 > 0, N2 \neq 0,$
$Y1 = N1 - 1, X1 = M1 - 1, Y2 = N2 - 1, X2 = M2 - 1, Z3 = Z2 + 1,$
$new1(M1, Y1, Z1, M2, Y2, Z2),\ new2(X1, Z1, A1, X2, Z3, A2)$

The basic idea of our pairing strategy is that the atoms that are paired together, having some of their arguments equal, have a somewhat synchronized behavior and this synchronization may determine, for the other arguments, the existence of simple relations that are easy to express in the theory of constraints one considers.

At this point of the application of the Predicate Pairing strategy we have that $\mathit{TransfCls} = Q \cup R \cup \{12, 13, 16, 18\}$, $\mathit{Defs} = \{11, 17\}$, and $\mathit{InCls} = \{17\}$.

- *Third iteration of the body of the while-loop of Predicate Pairing.*

Since $\mathit{InCls} = \{17\}$ and clause 17 has the atom $ack1$ that occurs in Q and the atom $ack2$ that occurs in R, we have to perform a new iteration of the body of the while-loop of the Predicate Pairing.

In clause 17 we unfold once the atom with predicate $ack1$ and the atom with predicate $ack2$, and we get:

19. $new2(M1, N1, A1, M2, N2, N2) \leftarrow M1 = M2, M1 \leq 0, A1 = N2 + 1$
20. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 = 0, N2 = 0,$
$X1 = M1 - 1, Y1 = 1, X2 = M2 - 1, Y2 = 1, ack1(X1, Y1, A1), ack2(X2, Y2, A2)$
21. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 = 0, N2 \neq 0,$
$X1 = M1 - 1, Y1 = 1, X2 = M2 - 1, Y2 = N2 - 1, Z3 = Z2 + 1,$
$ack1(X1, Y1, A1),\ ack2(M2, Y2, Z2),\ ack2(X2, Z3, A2)$
22. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 > 0, N2 = 0,$
$X1 = M1 - 1, Y1 = N1 - 1, X2 = M2 - 1, Y2 = 1,$
$ack1(M1, Y1, Z1), ack1(X1, Z1, A1), ack2(X2, Y2, A2)$
23. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 > 0, N2 \neq 0,$
$X1 = M1 - 1, Y1 = N2 - 1, X2 = M2 - 1, Y2 = N2 - 1, Z2 + 1 = Z3,$
$ack1(M1, Y1, Z1), ack1(X1, Z1, A1), ack2(M2, Y2, Z2), ack2(X2, Z3, A2)$

Clause 19 need not be folded. Clause 20 can be folded using definition 17 and we get:

24. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1 = M2, M1 > 0, N1 = 0, N2 = 0,$
$X1 = M1 - 1, Y1 = 1, X2 = M2 - 1, Y2 = 1, new2(X1, Y1, A1, X2, Y2, A2)$

In order to fold clause 21, first we select the two atoms with the predicates to be paired. We have that $ack1(X1, Y1, A1)$ shares one equality with $ack2(X2, Z3, A2)$, that is, $X1 = X2$ (this equality follows from $M1 = M2$), and shares no equalities with $ack2(M2, Y2, Z2)$. Hence we select the atoms $ack1(X1, Y1, A1)$ and $ack2(X2, Z3, A2)$

in the body of clause 21, and we fold that clause by using clause 17, thereby deriving the following clause:

25. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1{=}M2, M1{>}0, N1{=}0, N2 \neq 0,$
$X1{=}M1{-}1, Y1 = 1, X2{=}M2{-}1, Y2{=}N2{-}1, Z3{=}Z2{+}1,$
$new2(X1, Y1, A1, X2, Z3, A2), ack2(M2, Y2, Z2)$

By processing clauses 22 and 23 in a similar manner, we get:

26. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1{=}M2, M1{>}0, N1{>}0, N2{=}0,$
$X1{=}M1{-}1, Y1{=}N1{-}1, X2{=}M2{-}1, Y2{=}1,$
$new2(X1, Z1, A1, X2, Y2, A2), ack1(M1, Y1, Z1)$
27. $new2(M1, N1, A1, M2, N2, A2) \leftarrow M1{=}M2, M1{>}0, N1{>}0, N2{\neq}0,$
$X1{=}M1{-}1, Y1{=}N1{-}1, X2{=}M2{-}1, Y2{=}N2{-}1, Z3{=}Z2{+}1,$
$new2(M1, Y1, Z1, M2, Y2, Z2), new2(X1, Z1, A1, X2, Z3, A2)$

Since $InCls = \emptyset$ no new iteration of the body of the while-loop of the Predicate Pairing is required. Thus, the application of that strategy terminates. The resulting set of clauses is $TransfCls = Q \cup R \cup \{12, 13, 16, 18, 19, 24, 25, 26, 27\}$.

Now, the CHC solver Z3, when given as input the set *TransfCls* of clauses, constructs a LIA-definable model of *TransfCls*. In particular, it constructs a LIA-definable model of clause 12 by inferring that $new1(M1,N1,A1,M2,N2,A3)$ implies $A1 = A3+1$, which together with $A2 = A3+1$, implies $A1 = A2$, and hence the body of the clause is shown to be false.

By Theorem 2 the existence of a LIA-definable model for *TransfCls* entails that clauses 1–9 have a $\mathbb{Z}$-model, and this concludes the proof that programs `ackermann1` and `ackermann2` are equivalent.

We end this section by stating some results about the Predicate Pairing strategy. First, we have that Predicate Pairing always terminates because the number of new predicate definitions that can be introduced is bounded by the number k of different conjunctions of the form (e, A, B), where A and B are atoms whose predicates occur in $Q \cup R$, and e is a conjunction of equalities between variables in (A, B). Hence, the number of executions of the body of the while-loop of the Predicate Pairing strategy is at most k.

It is easy to see that the sequence of applications of the transformation rules performed by the Predicate Pairing strategy constructs a transformation sequence where every definition in *Defs* is unfolded at least once. Thus, from Theorem 2, which states the equivalence with respect to $\mathbb{D}$-satisfiability, and Theorem 4, which states the preservation of $\mathcal{A}$-definable models, we get the following result.

Theorem 6 (*Termination and soundness of the Predicate Pairing strategy*)
Let the sets $\{C_{init}\}$, Q, and R of clauses be the input of the Predicate Pairing strategy. Then the strategy terminates and returns a set *TransfCls* of clauses such that:
(i) $\{C_{init}\} \cup Q \cup R$ is $\mathbb{D}$-satisfiable iff *TransfCls* is $\mathbb{D}$-satisfiable, and
(ii) if $\{C_{init}\} \cup Q \cup R$ has an $\mathcal{A}$-definable model, then *TransfCls* has an $\mathcal{A}$-definable model.

Finally, note that the application of the Predicate Paring strategy may be iter-

ated, and hence, at the end of the transformation of a set of clauses, more than two predicates may turn out to be tupled together.

5 Case Studies: Relational Program Properties

In this section we illustrate the application of the Predicate Pairing strategy to some relevant classes of relational program properties. In particular, we have considered the following classes of properties: (i) the equivalence of programs implementing nonlinear and/or nested recursive functions, (ii.1) the injectivity of programs, (ii.2) the monotonicity of programs, (ii.3) the functional dependence of programs, (iii) non-interference of programs, (iv) equivalence of loop-optimized versions of programs with respect to the corresponding non-optimized versions, and (v) the equivalence of programs that manipulate integer arrays. We will consider these classes of properties in separate subsections.

Now we briefly show how to encode relational properties between the executions of two programs P and Q (De Angelis et al. 2016). We assume that the operational semantics of programs P and Q is represented by predicates $p(A, B)$ and $q(X, Y)$, where A and X represent (tuples of) input values, and B and Y represent (tuples of) output values, respectively. As already mentioned in the previous sections, the clauses defining p and q can be derived by specializing the interpreter of the imperative language with respect to the programs (De Angelis et al. 2017).

Let us now consider the relational property stating that, if the constraint $pre(A,B)$ holds before the execution of P and Q and the execution of these programs terminates, then the constraint $post(A, B, X, Y)$ holds after the execution. This property can be verified by testing the satisfiability of the CHC set consisting of the clauses defining predicates p and q, together with the following clause:

RP: $\mathit{false} \leftarrow \mathit{pre}(A, B),\ \mathit{notpost}(A, B, X, Y),\ p(A, B),\ q(X, Y)$

where $\mathit{notpost}(A, B, X, Y)$ is a constraint which is equivalent to the negation of $\mathit{post}(A, B, X, Y)$[1].

The application of our method based on the use of the Predicate Pairing strategy, is often crucial for solving satisfiability problems that encode relational program properties. In Section 6 we will discuss the results we have obtained in an extensive experimental evaluation that we have conducted.

5.1 Functions with Nonlinear and/or Nested Recursion

Similarly to what has been considered in Section 4, where we have presented two imperative programs implementing the Ackermann function specification and then proved their equivalence, in this section we consider various equivalence problems for pairs of imperative programs implementing some functional specifications. In each

[1] If the constraint language has no negation symbol, but the negation of a constraint is equivalent to a disjunction of constraints, as in the case of LIA, then the relational property can be encoded by a set of clauses.

pair one imperative program uses recursion only and the other one uses recursion and iteration.

The operational semantics of the two imperative programs is encoded using two distinct sets of CHCs, each defining a predicate for each program. The recursive structure of these predicate definitions mirrors the control flow of two imperative programs.

Let us consider two predicates $p1$ and $p2$ that encode the operational semantics of the two imperative programs, say $P1$ and $P2$, implementing a given function specification. The *equivalence* between $P1$ and $P2$ holds if, under some precondition on the input values, $p1$ and $p2$ define the same input/output relation. This property holds if the following clause, together with the set of clauses defining the predicates $p1$ and $p2$, is satisfiable:

EQ: $\mathit{false} \leftarrow c(X1),\ X1{=}X2,\ Y1{\neq}Y2,\ p1(X1,Y1),\ p2(X2,Y2)$

where: (i) $X1$, $X2$ represent tuples of input values, (ii) $Y1$, $Y2$ represent tuples of output values, and (iii) $c(X1)$ is a precondition on the input values. The reader may note that clause EQ is an instance of clause RP defining the general relational property. Note also that clause 9 in Section 4, encoding the equivalence relation between the two implementations of the Ackermann function, is an instance of EQ.

We have considered equivalence problems for imperative programs implementing *nonlinear* recursive functional specifications, that is, functional specifications with two or more recursive calls that depend on the same call (as in the case of the Fibonacci function). Also, several of these specifications have *nested recursions*, that is, they have recursive calls that are arguments of other recursive calls (as in the case of the Ackermann function), thus making the verification problem more challenging.

In particular, in our experiments we have considered the following specifications of variants of the Ackermann function[2]. (Here and in the other function definitions we assume that x, y, and z are non-negative integers.)

(1) Original version by W. Ackermann:
$A(0,y,z) = y{+}z, \quad A(1,y,0) = 0, \quad A(2,y,0) = 1,$
$A(x{+}3,y,0) = y, \quad A(x{+}1,y,z{+}1) = A(x,y,A(x{+}1,y,z))$

(2) Variant by H. Edelsbrunner:
$E(0,y,z) = y{+}z, \quad E(x{+}1,y,0) = 0, \quad E(x{+}1,y,1) = y,$
$E(x{+}1,y,z{+}2) = E(x,y,E(x{+}1,y,z{+}1))$

(3) Variant by R. Robinson (we have used this variant in Section 4):
$R(0,y) = y{+}1, \quad R(x{+}1,0) = R(x,1), \quad R(x{+}1,y{+}1) = R(x,R(x{+}1,y))$

(4) Variant by R. Péter:
$P(0,y) = 2y{+}1, \quad P(x+1,0) = P(x,1), \quad P(x{+}1,y{+}1) = P(x,P(x{+}1,y))$

Note that, these variants of the specification of the Ackermann function actually correspond to pairwise different functions. Indeed, we have that $A(2,y,0){\neq}E(2,y,0)$ for all $y{\geq}0$, and $R(0,y){\neq}P(0,y)$ for all $y{>}0$.

[2] `http://mrob.com/pub/math/ln-2deep.html`

Additionally, we have considered some other equivalence problems for pairs of imperative programs encoding the following functional specifications:

(5) a variant of the Sudan function:
$S(0, y, z) = y+z, \quad S(x+1, y, 0) = S(x, y+1, 0),$
$S(x+1, y, z+1) = S(x, S(x+1, y, z), y+S(x+1, y, z))$

(6) the B function:
$B(0, y) = y+1, \quad B(x+1, y) = B(x, B(x+1, y-1))$

(7) the G function:
$G(1, y) = y+3, \quad G(x+2, 0) = G(x+1, 1), \quad G(x+2, y+1) = G(x+1, G(x+2, y))$

(8) the McCarthy 91 function: $M(x) =$ if $x > 100$ then $x-10$ else $M(M(x+11))$

(9) the Dijkstra *fusc* function:
$fusc(0) = 0, \quad fusc(1) = 1, \quad fusc(2x) = fusc(x),$
$fusc(2x+1) = fusc(x+1) + fusc(x), \quad$ and

(10) a function that computes the minimum number of moves needed for the solution of the towers of Hanoi problem.

Our strategy turns out to be very effective in increasing the ability of the CHC solver to prove the satisfiability, or the unsatisfiability, of the clauses encoding the considered problems. As already mentioned, the main reason for this effectiveness is due to the fact that, by pairing together two atoms, the Predicate Pairing strategy often enables the discovery of relations between some of their arguments.

5.2 Monotonicity, Injectivity, and Functional Dependence

Some interesting classes of relational properties we have considered are those of monotonicity, injectivity, and functional dependency. These notions relate two different terminating executions of the same program on two distinct input values, say x and y, computing the output values, say m and n, respectively. The definition of these properties are derived in a straightforward manner from those of the mathematical functions.

In particular, monotonicity properties state that the application of the program on ordered input values produces ordered output values. For example, a typical monotonicity property is the following: if $x \leq y$, then $m \leq n$.

Injectivity properties state that any two executions of the same program on different inputs produce different outputs, that is, if $x \neq y$, then $m \neq n$.

Functional dependence properties state that the output of a program is a function of (a possibly proper subset of) its input values: for instance, if $x = y$, then $m = n$.

In particular, let us consider the following constrained Horn clauses encoding the operational semantics of a given imperative recursive program `Fib` that computes the Fibonacci numbers:

$fib(X, Y) \leftarrow X = 0, Y = 0$
$fib(X, Y) \leftarrow X = 1, Y = 1$
$fib(X, Y) \leftarrow X \geq 2, X1 = X-1, X2 = X-2, Y = Y1+Y2, \ fib(X1, Y1), \ fib(X2, Y2)$

where the first argument of *fib* encodes the input and the second argument of

fib encodes the output. Then, the above mentioned properties of monotonicity, injectivity, and functional dependence of the program `Fib` can be checked by testing the satisfiability of the following clauses:

$$\mathit{false} \leftarrow Y2 \geq Y1+1,\ X1 \geq X2,\ \mathit{fib}(X1,Y1),\ \mathit{fib}(X2,Y2) \quad \text{(Monotonicity)}$$
$$\mathit{false} \leftarrow Y1 = Y2,\ X1 \neq X2,\ \mathit{fib}(X1,Y1),\ \mathit{fib}(X2,Y2) \quad \text{(Injectivity)}$$
$$\mathit{false} \leftarrow Y1 \neq Y2,\ X1 = X2,\ \mathit{fib}(X1,Y1),\ \mathit{fib}(X2,Y2) \quad \text{(Functional Dependence)}$$

Note that the above clauses are all instances of clause RP encoding the general relational property.

Based on this example, the reader will not find it difficult to express monotonicity, injectivity, and functional dependence for other given imperative programs. We have successfully verified these properties for programs computing: (i) the sum of two numbers (by iterated increment), (ii) the product of two numbers (by iterated addition), and (iii) the square and the cube of a number (by iterated addition). We have also considered some more programs containing simple, sequential, or nested while-loops, possibly combined with conditionals.

5.3 Non-interference

Non-interference is a property that guarantees information-flow security. It can be viewed as a variant of the functional dependence property as we now indicate.

Let us consider an imperative program P whose variables are partitioned into a set of public variables (or low security variables) and a set of private variables (or high security variables). We say that P satisfies the non-interference property if any two terminating executions of P, starting with the same initial values of the public variables, but possibly with different values of the private variables, compute the same values of the public variables. Thus, if a program satisfies the non-interference property, an attacker cannot acquire information about the private variables by observing the input/output relation between the public variables, which are functionally dependent on the public input variables only.

To clarify the ideas, let us consider the following simple imperative program `HL`:

```
while (high >= 1) { high = high-1; low = high; }
```

where `low` is a public variable and `high` is a private variable. Program `HL` violates the non-interference property because there exist two different executions starting with identical values of the variable `low` and terminating in configurations having different values for variable `low`. Indeed, if initially we have that `high` is at least `1`, then the body of while-loop is executed and the final value of `low` will be `0`, otherwise the value of `low` is left unchanged.

The non-interference property for program P can be verified by checking the satisfiability of the following set of clauses:

$$\mathit{false} \leftarrow \mathit{OutL} \neq \mathit{OutL}1,\ L = L1,\ p(L,H,\mathit{OutL}),\ p(L1,H1,\mathit{OutL}1)$$
$$p(L,H,\mathit{OutL}) \leftarrow H < 1,\ \mathit{OutL} = L$$
$$p(L,H,\mathit{OutL}) \leftarrow H \geq 1,\ H1 = H-1,\ L1 = H1,\ p(L1,H1,\mathit{OutL})$$

where: (i) the predicate $p(L,H,\mathit{OutL})$ encodes the input/output relation among the variables of program P, (ii) the variables L and H encode the values of the variables

`low` and `high` at the beginning of the while-loop, and (iii) *OutL* encodes the value of the variable `low` at the end of the while-loop. Note that the set of clauses shown above is *unsatisfiable* because program P *violates* the non-interference property.

The reader may note that the first clause, encoding the non-interference property for program P, is an instance of clause RP defining the general relational property. The encoding of the non-interference property for other programs can be done in a similar way.

The following program `HL1` is representative of a class of programs for which we have successfully verified that the non-interference property holds:

```
low1 = low2;   low1 = low1 + f(high);   low1 = low1 - g(high,low1);
```

where: (i) `low1` and `low2` are public variables, `high` is a private variable, and (ii) `f` and `g` are two functions defined as follows:

```
int f(int m) {
  int i = 0, s = 0;
  while (i <= m) { s += i+m; i++; }
  return s;  }

int g(int m, int n) {
  int i = 0, s = 0;
  if (n <= m) { while ( i<= n) { s += i+m; i++; } } ;
  while (i <= m) { s += i+m; i++; }
  return s; }
```

Note that, in the program `HL1` the functions `f` and `g` compute the same value. This program `HL1` *does satisfy* the non-interference property, and thus the corresponding set of clauses is *satisfiable*. Indeed, in the program `HL1` the public variable `low1` is first incremented and then decremented by the same value, which, however, is computed by the distinct, yet equivalent functions `f` and `g`, which take the private variable `high` as input.

5.4 Loop Optimizations

Modern compilers often perform a series of optimizations for producing a new program that is semantically equivalent to an old program, but whose execution is faster, or requires less memory, or has lower energy consumption.

By applying our method we have successfully verified equivalence properties between some imperative programs and their optimized versions (Lopes and Monteiro 2016). The CHC encoding of program equivalence is the one defined by clause EQ in Section 5.1. For instance, we have proved the equivalence of the following program:

```
while (i < n) {
   if (n > 5) { a = a+n;   i = i+1; }
      else    { a = a+1;   i = i+1; }
}
```

and the one derived from it by the *loop unswitching* optimization:

```
if (n > 5) { while (i < n) { a = a+n;   i = i+1; } }
   else    { while (i < n) { a = a+1;   i = i+1; } }
```

where the conditional statement occurring in the while-loop is moved outside the loop, so that the evaluation of the conditional expression is performed only once, instead of being performed at each loop iteration.

We have also considered some specific instances of other equivalence problems relating original, non-optimized programs to new programs obtained by applying the following loop optimizations:

(i) *loop fission*, that splits the commands occurring in a loop in two blocks that are then executed by two consecutive, independent loops;

(ii) *loop fusion*, that merges the commands occurring in consecutive loops and executes them in a single loop;

(iii) *loop reversal*, that executes the commands occurring in a loop, in a new loop where the iteration proceeds in reversed order with respect to the order of the original loop;

(iv) *strength reduction*, that replaces iterated expensive computations in a loop by cheaper ones (for instance, replacing multiplication by a loop index with addition); and

(v) *code sinking*, that moves code occurring immediately before or after a loop inside the loop itself, possibly using conditionals for keeping the semantics of the program unaltered.

We have also considered other loop optimizations, including *loop tiling*, *loop aligning*, *loop pipelining* as well as other optimizations for removing redundant assignments, expression evaluations, and conditionals.

We are confident that our method of proving equivalence of programs can be extended for proving correctness of code optimizations at a schematic level (Leroy 2009; Lopes and Monteiro 2016), and not for some specific instances only. We leave this study for future research.

5.5 Array-manipulating Programs

We have applied our verification method to relational properties of imperative programs manipulating integer variables and integer arrays.

Let us first introduce some preliminary notions. An *integer array* a (or an *array*, for short) is a finite sequence of integers whose length, called the dimension of the array, is denoted $dim(a)$. An *atomic array constraint* is either $read(a, i, v)$, denoting that the i-th element of the array a is the integer v, or $write(a, i, v, b)$, denoting that, for $k = 1, \ldots, dim(a)$, if $k \neq i$, then the k-th element of a is equal to the k-th element of b, and if $k = i$, then the k-th element of b is the integer v.

The *read* and *write* constraints satisfy the following implicative axioms (Bradley et al. 2006), whose variables are assumed to be universally quantified at the front:

$$\big(I = J,\ read(A, I, U),\quad read(A, J, V)\big) \rightarrow U = V \qquad (\textit{array congruence})$$
$$\big(I = J,\ write(A, I, U, B),\ read(B, J, V)\big) \rightarrow U = V \qquad (\textit{read-over-write})$$
$$\big(I \neq J,\ write(A, I, U, B),\ read(B, J, V)\big) \rightarrow read(A, J, V) \qquad (\textit{read-over-write})$$

For example, the operational semantics of the following imperative program which acts on the array `a`:

```
i = 1;   a[0] = 3;
while ( i < n ) { a[i] = a[i-1]+2;  i++; }
```

can be represented by the following set of CHCs:

$prog(N, A1, A3) \leftarrow I=1,\ K=0,\ U=3,\ write(A1, K, U, A2),\ loop(N, A2, I, A3)$
$loop(N, A1, I, A3) \leftarrow I+1 \leq N,\ J=I-1,\ read(A1, J, U),$
$\qquad V=U+2,\ write(A1, I, V, A2),\ I1=I+1,\ loop(N, A2, I1, A3)$
$loop(N, A, I, A) \leftarrow I \geq N$

In these clauses: (i) the predicate *loop(N, A_in, I, A_out)* encodes the while-loop, (ii) its arguments N, *A_in*, and I encode the values of variables `n`, `a`, and `i`, respectively, at loop entry, and (iii) *A_out* encodes the value of `a` at loop exit.

Now we show an example of an equivalence property between array manipulating programs that has been proved by using our method based on the Predicate Pairing strategy. Let us consider the programs `P1` and `P2` shown in Table 1, where program `P2` is obtained from program `P1` by applying the loop-pipelining optimization, a commonly used technique for enabling instruction-level parallelism at the hardware level.

Program `P1`	Program `P2`

	`i = 0;`
	`a[0]++;`
	`b[0] += a[0];`
	`a[1]++;`
	`while ( i < n-2 ) {`
	`  a[i+2]++;`
	`  b[i+1] += a[i+1];`
	`  c[i] += b[i];`
	`  i++;`
	`}`
	`c[i] += b[i];`
	`b[i+1] += a[i+1];`
	`c[i+1] += b[i+1];`

`i=0;`
`while (i < n) {`
`  a[i]++;`
`  b[i] += a[i];`
`  c[i] += b[i];`
`  i++;`
`}`	

Table 1. The source program `P1` (left) and the optimized program `P2` obtained by applying the loop pipelining transformation (right).

The equivalence between programs `P1` and `P2` with respect to the output array `c`, is expressed by the following clause F (which is an instance of clause EQ of Section 5.1):

F: $false \leftarrow X \neq Y,\ N \geq 1,\ J \geq 0,\ J \leq N-1,\ read(C1, J, X), read(C2, J, Y),$
$\qquad new11(N, A, C1),\ new21(N, A, C2)$

where $new11(N, A, C1)$ represents the input/output relation of the source program `P1`, and in particular, N is the value of the integer variable `n`, A is the value of the array `a` at the beginning of program execution, and $C1$ is the value of the ar-

ray `c` at the end of program execution. Similarly, *new*21 represents the input/output relation of the optimized program `P2`.

Thus, by proving the satisfiability of the set of clauses consisting of F together with the clauses defining *new*11 and *new*21, we have been able to prove that programs `P1` and `P2` produce identical values for the array `c` as output, when provided with identical values for the array `a` as input.

6 Experimental Evaluation

In this section we present the experimental evaluation we have performed for assessing the effectiveness of the Predicate Pairing strategy (PP strategy, for short) presented in Section 4.

Implementation. We have implemented Algorithm 1 by using the VeriMAP system (De Angelis et al. 2014b), which is a tool for software model checking based on transformation techniques for CHCs. Then we have used the SMT solver Z3 (de Moura and Bjørner 2008) for checking the satisfiability of the clauses generated by VeriMAP. In particular, we have used Z3 version 4.5.0 with the Duality fixed-point engine (McMillan and Rybalchenko 2013), which provides support for constraints defined on linear integers and integer arrays.

Our prototype implementation consists of two components: (1) a module that realizes Algorithm 1, and (2) a module that translates the generated CHCs into the SMT-LIB format which is the format accepted by Z3 (see Figure 3). The VeriMAP system also provides a front-end module (T) that takes a pair of C programs, together with a relational property to be verified, and translates them into the CHCs that encode the verification problem (De Angelis et al. 2016).

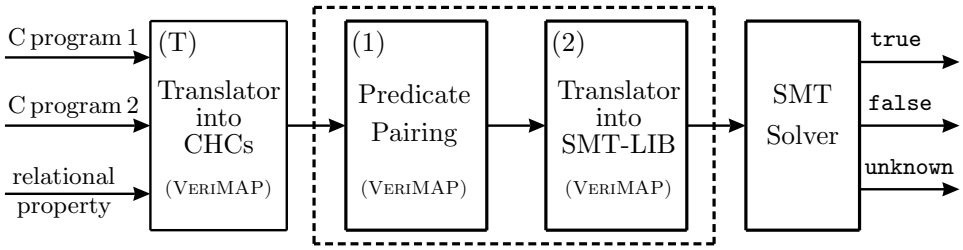

Figure 3. The Verification System: (1) the Predicate Pairing module, implementing Algorithm 1, and (2) the Translator to SMT-LIB module.

Verification problems. We have considered a benchmark[3] of 163 sets of CHCs (153 of which are satisfiable and the other 10 are unsatisfiable), representing veri-

[3] The benchmark suite can be found at `http://map.uniroma2.it/predicate-pairing`. The suite includes the CHCs that encode the relational verification problems (both in Prolog and SMT-LIB format), the transformation logs, the transformed CHCs, the statistics on the transformations and on the Z3 satisfiability checks. For the verification problems generated from C programs, the suite also contains the C sources.

fication problems (all acting on integers and 14 of them acting on integer arrays), which refer to relational properties of small, yet non-trivial, imperative programs mostly taken from the literature (Barthe et al. 2011; Benton 2004; Felsing et al. 2014; De Angelis et al. 2015a; De Angelis et al. 2016; Lopes and Monteiro 2016).

We have considered the following categories of relational verification problems. (1) The category NLIN, which refers to equivalence properties between programs implementing functions with nonlinear and/or nested recursion. For instance, we have verified the equivalence of two programs for computing the Ackermann function (see the running example presented in Section 4 and the examples of Section 5.1). (2)–(6) The categories MON, INJ, FUN, NINT, and LOPT, which refer to monotonicity, injectivity, functional dependence, non-interference, and loop optimizations problems, respectively (see Section 5). (7) The category ITE, which refers to equivalence properties and inequality properties relating two iterative programs acting on integers (by an inequality property we mean that the values computed by the programs are related by $\leq$). (8) The category ARR, which refers to equivalence and inequality properties between iterative programs acting on integer arrays. (9) The category REC, which refers to equivalence properties between recursive programs. (10) The category I-R, which refers to equivalence properties between an iterative program and a (non-tail) recursive program. For example, we have verified the equivalence of the iterative and recursive versions of programs for computing: (i) the greatest common divisor of two integers, and (ii) the m-th triangular number, that is, $\sum_{x=1}^{m} x$. (11) The category COMP, which refers to equivalence and inequality properties between programs that contain compositions of different numbers of loops acting on integers (3 problems) and integer arrays (7 problems). (12) The category PCOR, which refers to partial correctness properties of an iterative program with respect to a recursive functional postcondition (De Angelis et al. 2015a).

Note that in our benchmark set, 31 problems (belonging to the categories NLIN, NINT, and COMP) are encoded by sets of clauses that include nonlinear clauses, besides the ones that encode the relational properties, which are always nonlinear (see Sections 4 and 5 for some examples).

Technical resources. The experimental evaluation has been performed on a single core of an Intel Core Duo E7300 2.66GHz processor with 4GB of memory running Ubuntu. For all problems we have set the timeout limit of 300 seconds.

Experimental processes. We have considered the following two experimental processes. (E1) The first experimental process consists in running Z3 for checking the satisfiability of the original sets of CHCs that encode the verification problems. (E2) The second experimental process consists in running Algorithm 1 on the original sets of CHCs for each verification problem, and then running Z3 for checking the satisfiability of the derived CHCs. In this second process the PP strategy has been iterated (see the end of Section 4), if more than one pair of atoms was present in the bodies of the original sets of clauses. In particular, PP has been iterated for 24 sets of CHCs, in total, belonging to categories NINT, ARR, COMP, and PCOR.

Results. The results of the experimental evaluation are summarized in Table 2. The times reported are the CPU seconds spent in user mode and kernel mode by

Problems		Z3 before PP		PP	Z3 after PP	
Category	P	S_1	T_1	T_{PP}	S_2	T_2
(1) NLIN	13	4	16.11	25.80	13	13.12
(2) MON	18	1	1.04	2.27	12	3.72
(3) INJ	11	0	–	1.36	8	1.39
(4) FUN	7	4	1.39	1.24	7	1.48
(5) NINT	18	3	0.27	55.80	17	41.33
(6) LOPT	20	2	4.83	2.98	15	10.71
(7) ITE	22	5	26.67	4.53	18	17.01
(8) ARR	6	1	7.45	2.04	5	3.25
(9) REC	15	6	2.89	1.50	13	4.28
(10) I-R	4	0	–	0.65	3	1.02
(11) COMP	10	0	–	16.35	7	6.46
(12) PCOR	19	5	83.93	17.84	17	17.65
Total number	163	31	144.58	132.36	135	121.42
Average Time			4.66	0.81		0.90

Table 2. The first two columns report the names of the problem categories and the number P of problems in each category, respectively. Columns S_1 and S_2 report the number of verification problems solved by Z3 before and after the application of the PP strategy, respectively. Columns T_1 and T_2 report the time taken by Z3 to solve the problems reported in columns S_1 and S_2, respectively. Column T_{PP} reports the time taken by VeriMAP to apply the PP strategy on the P problems. Times are in seconds. The timeout is of 300 seconds. No timeout occurred during the application of the PP strategy.

(i) VeriMAP for transforming the clauses, and (ii) Z3 for checking their satisfiability. The times required for translating the CHCs into the SMT-LIB format (this translation is necessary for running Z3) are: 6.39 seconds for the first experimental process E1, and 22.59 seconds for the second experimental process E2. Those translation times are not very high with respect to the solving times which are 144.58 seconds and 121.42 seconds, respectively.

As expected, the use of the PP strategy significantly increases the number of problems that are solved by Z3. In particular, the number of solved problems increases from 31 (Total of Column S_1) to 135 (Total of Column S_2). Note, however, the PP strategy can derive a set of clauses which is larger than the original set. In our benchmark we have observed that size increases of about 2.16 times on average.

Note also that the PP strategy increases the efficiency of the satisfiability check. Indeed, the average time taken to run PP and then Z3 (1.88 seconds) is lower

than the average time taken to run Z3 on the original set of clauses (4.66 seconds). Although we have proved that the application of the PP strategy preserves all LIA-definable models, in our experiments we have found three problems which Z3 was able to solve before the application of the PP strategy (i.e., in the first experimental process), and it was no longer able to solve, within the given time limit, after the application of the PP strategy (i.e, in the second experimental process). This phenomenon may be due to the fact that the termination of the algorithms implemented by the Z3 solver is sensitive to the syntactic form of the input clauses, and the PP strategy modifies that form. Further work is needed to improve the termination behavior of the solver.

Finally, we would like to point out that the application of the PP strategy does not decrease the efficiency of the whole verification process. If we consider only the 28 problems for which Z3 is able to solve both before and after the application of the PP strategy, the average time taken to run PP and then Z3 (1.58 seconds) is slightly lower than the time taken by Z3 alone (1.77 seconds).

7 Related Work and Conclusions

The basic idea behind the Predicate Pairing transformation strategy for constrained Horn clauses is that, by finding a recursive definition of a predicate denoting the conjunction of two atoms, it is often possible to infer relations among the variables occurring in the two atoms which would have been impossible to discover by considering each atom separately.

Techniques for transforming logic programs by deriving new predicates defined in terms of conjunctions of atoms have been largely studied. Let us recall, for instance, the well-known *tupling* transformation strategy (Pettorossi and Proietti 1994) and the *conjunctive partial deduction* technique for logic program specialization (De Schreye et al. 1999). The main objective of tupling and conjunctive partial deduction is the derivation of more efficient logic programs by avoiding multiple traversals of data structures and repeated evaluations of predicate calls, and by producing specialized program versions.

Thus, Predicate Pairing shares with tupling and conjunctive partial deduction the idea of promoting a conjunction of atoms to a new predicate. However, in this paper we have shown that the application of this idea to constrained Horn clauses can also play a key role in improving the effectiveness of CHC solvers for proving properties of imperative programs, besides the optimization of the execution of logic programs, which is the objective of tupling and conjunctive partial deduction. This is the case especially when the CHC solvers are required to test the satisfiability of clauses that encode relational program properties, that is, properties that relate two programs, or two executions of the same program. Indeed, as shown by many examples considered in this paper, state-of-the-art solvers often fail to prove the satisfiability of sets of clauses encoding relational properties because they can only infer relations among the variables of individual atoms.

We have considered CHC solvers that prove satisfiability by using predicate abstraction, that is, by looking for models that are definable in a specific class $\mathcal{A}$ of

constraints (Bjørner et al. 2015). We have shown that, in principle, the Predicate Pairing strategy cannot worsen the effectiveness of the CHC solver. Indeed, we have proved a very general result concerning the unfold/fold transformation rules used by the strategy: if a set of clauses is transformed by applying the unfold/fold rules, and the original set of clauses has an $\mathcal{A}$-definable model, then also the transformed set of clauses has an $\mathcal{A}$-definable model. Thus, if the CHC solver is able to find an $\mathcal{A}$-definable model whenever it exists (and this is indeed possible if the validity problem for the constraints in $\mathcal{A}$ is decidable), then every set of clauses that can be proved satisfiable by the solver before the transformation, will also be proved satisfiable by the solver after the transformation. We have shown that, in practice, for the Z3 solver this property is guaranteed with very few exceptions.

We have also given some restrictions on the use of the rules that guarantee that the converse of the above property holds, that is: if a set of clauses is transformed by applying the unfold/fold rules, and the transformed set of clauses has an $\mathcal{A}$-definable model, then also the original set of clauses has an $\mathcal{A}$-definable model. However, this property is not always desirable. Indeed, the fact that in some cases Predicate Pairing is able to transform (satisfiable) clauses that do not have an $\mathcal{A}$-definable model into clauses that have an $\mathcal{A}$-definable model, may be a great advantage. Indeed, this means that while a CHC solver that looks for $\mathcal{A}$-definable models is not able to prove the satisfiability of the original clauses, the same solver may be able to prove the satisfiability of the transformed clauses.

The study of the properties that relate unfold/fold transformations and the existence of $\mathcal{A}$-definable models is not present in the literature on tupling and conjunctive partial deduction.

Then, we have presented an algorithm that implements the Predicate Pairing strategy. One of the novel points of this algorithm with respect to tupling and conjunctive partial deduction is that it realizes a heuristic to choose the appropriate atoms to be paired together in a new predicate definition, by maximizing the number of equality constraints that relate the variables occurring in a pair of atoms. This heuristic is crucially needed when dealing with nonlinear clauses that, by unfolding, may generate clauses with more than two atoms in their body. We have implemented our algorithm on the VeriMAP transformation and verification system (De Angelis et al. 2014b), and we have evaluated its effectiveness on a benchmark of over 160 problems encoding relational properties of small, yet non-trivial C-like programs. The properties were of various kinds, including equivalence, injectivity, functional dependence, and non-interference (a property of interest for enforcing software security). The results show that the use of Predicate Pairing as a preprocessor greatly improves the ability of the Z3 CHC solver (with the Duality fixed-point computation engine (McMillan and Rybalchenko 2013)) to prove satisfiability.

Several transformation-based techniques for constrained Horn clauses, or constraint logic programs, have been proposed as a means of facilitating program verification. Many of them are non-conjunctive specialization techniques, which work by propagating the constraints occurring in the goal, thereby producing clauses with strengthened constraints in their bodies (Albert et al. 2007; De Angelis et al.

2014a; De Angelis et al. 2017; De Angelis et al. 2015a; Kafle and Gallagher 2017a; Kafle and Gallagher 2017b; Méndez-Lojo et al. 2008; Peralta et al. 1998). Even if specialization techniques have been shown to be very successful, in most cases they cannot achieve the same effect as Predicate Pairing. Indeed, as already mentioned, Predicate Pairing works by introducing new predicates corresponding to conjunctions of old predicates, whereas non-conjunctive specialization can only introduce new predicates that correspond to instances of old predicates. We have experimentally checked that most of the problems considered in Section 6 cannot be solved via (non-conjunctive) specialization alone. Due to lack of space we have not reported these results.

The query-answer transformation (and variants thereof) is another pre-processing technique that is sometimes applied before performing satisfiability tests using CHC solvers (De Angelis et al. 2014a; Kafle and Gallagher 2017a; Kafle and Gallagher 2017b). The aim of this transformation is to simulate the top-down, goal oriented evaluation of the clauses in a bottom-up framework. The results we have presented here, showing that Predicate Pairing is able to transform clauses without an $\mathcal{A}$-definable model into clauses with an $\mathcal{A}$-definable model, are independent of the evaluation strategy adopted by the CHC solvers, and hence the query-answer transformation and the Predicate Pairing strategy should be viewed as orthogonal techniques.

Predicate Pairing is an extension of the *Linearization* transformation, whose objective is to transform a set of linear clauses (that is, clauses with at most one atom in their body) together with a nonlinear goal, into a set of linear clauses and linear goals (De Angelis et al. 2015b). The Predicate Pairing strategy does not need any linearity assumption, and indeed in Sections 4, 5, and 6 we have shown that this strategy can solve several verification problems encoded by sets of nonlinear clauses. It has also been shown that Linearization preserves the existence of LIA-definable models (De Angelis et al. 2015b). Here we have generalized this result by proving that the application of the unfold/fold transformation rules, independently of the strategy, preserves the existence of $\mathcal{A}$-definable models, for any class $\mathcal{A}$ of constraints.

The Predicate Pairing algorithm presented here is an improvement of the one reported in previous work presented at the SAS Symposium (De Angelis et al. 2016). Indeed, as already mentioned, here we use an equality-based heuristic to choose the appropriate atoms to be paired together, and this technique has been shown very effective in practice for handling nonlinear clauses. Also the case studies and the benchmark set we consider in the present paper are much larger, and include verification problems such as non-interference, correctness of loop optimizations, and equivalence of nonlinear recursive programs that have not been considered in our SAS paper (De Angelis et al. 2016). Moreover, in the present paper we include general results concerning the preservation of $\mathcal{A}$-models.

Bjørner et al. have shown that unfolding preserves $\mathcal{A}$-definable models provided that the set $\mathcal{A}$ of constraints admits Craig interpolation (Bjørner et al. 2015). In the present paper we have generalized this result by considering also other transformations, and in particular folding, and we dropped the assumption about Craig

interpolation. Moreover, we assume that $\mathcal{A}$ is a subset of the set $\mathcal{C}$ of constraints over which the clauses are defined, while Bjørner et al. take $\mathcal{A}$ to be equal to $\mathcal{C}$. Our generalization is significant because sometimes CHC solvers that make use of predicate abstraction look for models defined in subsets of the constraints used for the clauses, such as the popular domain of the octagons (Miné 2006).

The problem of verifying relational properties is very relevant in the context of software engineering. Indeed, during software development it is often the case that one modifies the program text, and hence needs a proof that the semantics of the new program version has some specified relation to the semantics of the old version. This kind of proof is sometimes called *regression verification* (Godlin and Strichman 2008).

Several logics and methods have been presented in the literature for reasoning about various relational program properties. A Hoare-like axiomatization of relational reasoning for simple while programs has been proposed by Benton (Benton 2004), who however does not present any technique for the automation of the proofs.

Program equivalence is one of the relational properties that have been extensively studied in the past, and still receives remarkable attention in recent work (Barthe et al. 2011; Chaki et al. 2012; Ciobâcă et al. 2014; Fedyukovich et al. 2016; Felsing et al. 2014; Godlin and Strichman 2008; Lopes and Monteiro 2016; Strichman and Veitsman 2016; Verdoolaege et al. 2012; Zaks and Pnueli 2008). A fruitful idea for easing the problem of proving program equivalence is to reduce it to a standard verification task by using some *composition* operator between imperative programs (Barthe et al. 2011; Lahiri et al. 2013; Zaks and Pnueli 2008). The application of these operators requires human ingenuity, and it is still necessary to provide the suitable invariants to be used by the program verifier.

A method for reusing available verification techniques to prove program equivalence is proposed by Ganty et al. (Ganty et al. 2013), who identify a class of recursive programs for which it is possible to precisely compute the so called summaries. This method can be used to reduce the problem of checking the equivalence of two recursive programs to the problem of checking the equivalence of their summaries.

Lopes and Monteiro proposed a different method for proving program equivalence that is based on the computation of precise (that is, not over-approximated) summaries (Lopes and Monteiro 2016). This method considers programs over the integers and is based on a transformation into integer (possibly nonlinear) polynomials. The equivalence checking algorithm then works on loop-free programs. This method has been applied to prove the correctness of several loop optimizations. As shown in Section 5, Predicate Pairing is able to prove similar correctness properties, by avoiding the use of nonlinear arithmetic.

Felsing et al. propose a technique for proving relational properties of imperative programs, which is based on a translation of special purpose proof rules into constrained Horn clauses (Felsing et al. 2014). The satisfiability of these clauses is then checked by state-of-the-art CHC solvers. The main difference between our approach and the one of Felsing et al. is that we generate a translation of the relational properties into sets of CHCs starting from the semantics of the imperative language, and hence we do not need any special purpose proof rule that depends on the pro-

gramming language and the class of properties under consideration. Instead, we use language independent transformation rules for CHCs. In particular, unlike Felsing et al., by using our approach we are able to verify relations between programs that have different structure, because the transformation rules are independent of the syntax of the source programs.

In conclusion, we would like to stress that our work confirms once again the great advantages offered by the program verification approach based on the use of constrained Horn clauses. Indeed, by reducing the problem of verifying properties of programs in a given language to the problem of reasoning with constrained Horn clauses, we are able to use general purpose techniques and very effective tools developed over the last four decades in the fields of logic programming, constraint-based reasoning, and automated theorem proving. In this way, we get verification methods with a very high level of flexibility and parametricity with respect to the language in which programs are written.

8 Acknowledgments

We warmly thank the anonymous referees for their very helpful comments and criticism. This work has been partially supported by the National Group of Computing Science (GNCS-INDAM). E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at CNR-IASI, Roma, Italy.

References

Albert, E., Gómez-Zamalloa, M., Hubert, L., and Puebla, G. 2007. Verification of Java bytecode using analysis and transformation of logic programs. In *Practical Aspects of Declarative Languages*, M. Hanus, Ed., Lecture Notes in Computer Science 4354. Springer, 124–139.

Barthe, G., Crespo, J. M., and Kunz, C. 2011. Relational verification using product programs. In *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*. Lecture Notes in Computer Science 6664. Springer, 200–214.

Benoy, F. and King, A. 1997. Inferring argument size relationships with CLP(R). In *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996*, J. P. Gallagher, Ed., Lecture Notes in Computer Science 1207. Springer, 204–223.

Benton, N. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*. ACM, 14–25.

Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, L. D. Beklemishev, A. Blass, N. Dershowitz, B. Finkbeiner, and W. Schulte, Eds., Lecture Notes in Computer Science 9300. Springer, Switzerland, 24–51.

Bradley, A. R., Manna, Z., and Sipma, H. B. 2006. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and*

Abstract Interpretation. VMCAI '06. Lecture Notes in Computer Science, vol. 3855. Springer, 427–442.

Chaki, S., Gurfinkel, A., and Strichman, O. 2012. Regression verification for multi-threaded programs. In *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, V. Kuncak and A. Rybalchenko, Eds., Lecture Notes in Computer Science 7148. Springer, 119–135.

Ciobâcă, S., Lucanu, D., Rusu, V., and Rosu, G. 2014. A language-independent proof system for mutual program equivalence. In *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, S. Merz and J. Pang, Eds., Lecture Notes in Computer Science 8829. Springer, 75–90.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 4th ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*. ACM, 238–252.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014a. Program verification via iterated specialization. *Science of Computer Programming 95, Part 2*, 149–175. Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2014b. VeriMAP: A tool for verifying programs through transformations. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '14*. Lecture Notes in Computer Science 8413. Springer, 568–574. Available at: http://www.map.uniroma2.it/VeriMAP.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2015a. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming 15,* 4–5, 635–650.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2015b. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae 140,* 3–4, 329–355.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2016. Relational verification through Horn clause transformation. In *Proceedings of the 23rd International Symposium on Static Analysis, SAS 2016, Edinburgh, UK, September 8-10, 2016*, X. Rival, Ed., Lecture Notes in Computer Science 9837. Springer, 147–169.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2017. Semantics-based generation of verification conditions via program specialization. *Science of Computer Programming 147,* 3–4, 78–108.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*. Lecture Notes in Computer Science 4963. Springer, 337–340.

De Schreye, D., Glück, R., Jørgensen, J., Leuschel, M., Martens, B., and Sørensen, M. H. 1999. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming 41,* 2–3, 231–277.

Debray, S. K. and Ramakrishnan, R. 1994. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming 18*, 149–176.

Etalle, S. and Gabbrielli, M. 1996. Transformations of CLP modules. *Theoretical Computer Science 166*, 101–146.

Fedyukovich, G., Gurfinkel, A., and Sharygina, N. 2016. Property directed equival-

ence via abstract simulation. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, Canada, July 17-23, 2016, Proceedings, Part II*, S. Chaudhuri and A. Farzan, Eds., Lecture Notes in Computer Science, vol. 7792. Springer International Publishing.

Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., and Ulbrich, M. 2014. Automating regression verification. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, September 15-19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds., 349–360.

Ganty, P., Iosif, R., and Konečný, F. 2013. Underapproximation of procedure summaries for integer programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, N. Piterman and Scott A. Smolka, Eds., Lecture Notes in Computer Science 7795, Springer, 245–259.

Godlin, B. and Strichman, O. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica 45,* 6, 403–439.

Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Synthesizing software verifiers from proof rules. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*. 405–416.

Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. 2015. The SeaHorn verification framework. In *Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015*. Lecture Notes in Computer Science 9206. Springer, 343–361.

Hoare, C. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10 (October), 576–580.

Hoder, K., Bjørner, N., and de Moura, L. M. 2011. μZ– An efficient engine for fixed points with constraints. In *Computer Aided Verification, 23rd International Conference, CAV '11, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, G. Gopalakrishnan and S. Qadeer, Eds., Lecture Notes in Computer Science 6806. Springer, 457–462.

Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., and Rümmer, P. 2012. A verification toolkit for numerical transition systems. In *FM '12: Formal Methods, 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, D. Giannakopoulou and D. Méry, Eds., Lecture Notes in Computer Science 7436. Springer, 247–251.

Jaffar, J. and Maher, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19/20*, 503–581.

Jaffar, J., Santosa, A., and Voicu, R. 2009. An interpolation method for CLP traversal. In *Principles and Practice of Constraint Programming, CP '09*, I. Gent, Ed., Lecture Notes in Computer Science 5732. Springer, 454–469.

Kafle, B. and Gallagher, J. P. 2017a. Constraint specialisation in Horn clause verification. *Science of Computer Programming 137,* 125–140.

Kafle, B. and Gallagher, J. P. 2017b. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures 47*, 2–18.

Kafle, B., Gallagher, J. P., and Morales, J. F. 2016. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*. Lecture Notes in Computer Science 9779. Springer, 261–268.

Lahiri, S. K., McMillan, K. L., Sharma, R., and Hawblitzel, C. 2013. Differential

assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, B. Meyer, L. Baresi, and M. Mezini, Eds., ACM, 345–355.

Leroy, X. 2009. Formal verification of a realistic compiler. *Communications of the ACM 52,* 7, 107–115.

Leuschel, M. and Bruynooghe, M. 2002. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming 2,* 4&5, 461–515.

Lloyd, J. W. 1987. *Foundations of Logic Programming.* Springer-Verlag, Berlin. Second Edition.

Lopes, N. P. and Monteiro, J. 2016. Automatic equivalence checking of programs with uninterpreted functions and integer arithmetic. *International Journal on Software Tools for Technology Transfer 18,* 4, 359–374.

McMillan, K. L. and Rybalchenko, A. 2013. Solving constrained Horn clauses using interpolation. MSR Technical Report 2013-6, Microsoft Report.

Mendelson, E. 1997. *Introduction to Mathematical Logic.* Chapman & Hall, London, UK. Fourth Edition.

Méndez-Lojo, M., Navas, J. A., and Hermenegildo, M. V. 2008. A flexible, (C)LP-based approach to the analysis of object-oriented programs. In *17th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR '07, Kongens Lyngby, Denmark, August 23-24, 2007.* Lecture Notes in Computer Science 4915. Springer, 154–168.

Miné, A. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation 19,* 1, 31–100.

Peralta, J. C., Gallagher, J. P., and Saglam, H. 1998. Analysis of imperative programs through analysis of constraint logic programs. In *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, G. Levi, Ed., Lecture Notes in Computer Science 1503. Springer, 246–261.

Pettorossi, A. and Proietti, M. 1994. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming 19,20*, 261–320.

Podelski, A. and Rybalchenko, A. 2007. ARMC: The logical choice for software model checking with abstraction refinement. In *Practical Aspects of Declarative Languages, PADL '07*, M. Hanus, Ed., Lecture Notes in Computer Science 4354. Springer, 245–259.

Rümmer, P., Hojjat, H., and Kuncak, V. 2013. Disjunctive interpolants for Horn clause verification. In *Proceedings of the 25th International Conference on Computer Aided Verification, CAV '13, Saint Petersburg, Russia, July 13-19, 2013*, N. Sharygina and H. Veith, Eds., Lecture Notes in Computer Science 8044. Springer, 347–363.

Strichman, O. and Veitsman, M. 2016. Regression verification for unbalanced recursive functions. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings.* Lecture Notes in Computer Science, vol. 9995. Springer International Publishing, 645–658.

Tamaki, H. and Sato, T. 1984. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, S.-Å. Tärnlund, Ed., Uppsala University, Uppsala, Sweden, 127–138.

Verdoolaege, S., Janssens, G., and Bruynooghe, M. 2012. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst. 34,* 3, 11.

Zaks, A. and Pnueli, A. 2008. CoVaC: Compiler validation by program analysis of the cross-product. In *Proceedings of the 15th International Symposium on Formal Methods (FM 2008), Turku, Finland, May 26-30, 2008*, J. Cuéllar, T. S. E. Maibaum, and K. Sere, Eds., Lecture Notes in Computer Science 5014. Springer, 35–51.

Appendix

Proof of Theorem 3

Proof
Let us assume that there exists an $\mathcal{A}$-definable model Σ of P_i that is tight on $Defs_i$. We will construct an $\mathcal{A}$-definable model Σ' of P_{i+1} that is tight on $Defs_{i+1}$. The proof proceeds by cases on the transformation rule applied to derive P_{i+1} from P_i.

(*Case* R1) Suppose that P_{i+1} is derived from P_i by applying the definition rule. Thus, $P_{i+1} = P_i \cup \{D\}$ and $Defs_{i+1} = Defs_i \cup \{D\}$, where D is the clause $newp(X_1, \ldots, X_k) \leftarrow c, G$, and the following conditions hold: (i) $newp$ is a new predicate symbol, (ii) $c \in \mathcal{A}$, (iii) all predicates occurring in G also occur in P_0, and (iv) $X_1, \ldots, X_k$ are distinct variables occurring free in (c, G).

Let Σ' be a symbolic interpretation that is equal to Σ for all atoms whose predicate is different from $newp$, and $\Sigma'(newp(X_1, \ldots, X_k)) = \exists Y_1 \ldots \exists Y_m(c \wedge \Sigma(G))$ where $\{Y_1, \ldots, Y_m\} = Fvars(c \wedge \Sigma(G)) \backslash \{X_1, \ldots, X_k\}$.

Now we have that Σ' is an $\mathcal{A}$-definable model of P_{i+1}, as the following two points hold:

Point (i): Σ' is an $\mathcal{A}$-definable model of P_i because Σ' is equal to Σ for all atoms whose predicates occur in P_i, and

Point (ii): Σ' is an $\mathcal{A}$-definable model of D, that is,
$\mathbb{D} \models \forall(c \wedge \Sigma'(G) \rightarrow \Sigma'(newp(X_1, \ldots, X_k)))$.

Point (ii) is shown as follows. Since $Y_1, \ldots, Y_m$ do not occur free in the formula $\Sigma'(newp(X_1, \ldots, X_k))$ and $newp$ does not occur in G,
$\mathbb{D} \models \forall(c \wedge \Sigma'(G) \rightarrow \Sigma'(newp(X_1, \ldots, X_k)))$
iff $\mathbb{D} \models \forall(\exists Y_1 \ldots \exists Y_m(c \wedge \Sigma(G)) \rightarrow \Sigma'(newp(X_1, \ldots, X_k)))$
and the latter implication holds by the definition of Σ'. Moreover, from the definition of Σ' and from the hypothesis that Σ is tight on $Defs_i$, it follows immediately that Σ' is tight on $Defs_{i+1}$.

(*Case* R2) Suppose that P_{i+1} is derived from P_i by applying the unfolding rule. Thus, $P_{i+1} = (P_i \backslash \{C\}) \cup \{H \leftarrow c, c_j, G_1, B_j, G_2 \mid j = 1, \ldots, m\}$, where C is the clause $H \leftarrow c, G_1, p(X_1, \ldots, X_k), G_2$ in P_i and $\{p(X_1, \ldots, X_k) \leftarrow c_j, B_j \mid j = 1, \ldots, m\}$ is the set of clauses in P_i whose head predicate is p.

Now we show that Σ is an $\mathcal{A}$-definable model of P_{i+1} that is tight on $Defs_{i+1}$. By the hypothesis that Σ is an $\mathcal{A}$-definable model of P_i we have that
$\mathbb{D} \models \forall(c \wedge \Sigma(G_1) \wedge \Sigma(p(X_1, \ldots, X_k)) \wedge \Sigma(G_2) \rightarrow \Sigma(H))$
and, for $j = 1, \ldots, m$,
$\mathbb{D} \models \forall(c_j \wedge \Sigma(B_j) \rightarrow \Sigma(p(X_1, \ldots, X_k)))$.
Then, for $j = 1, \ldots, m$,
$\mathbb{D} \models \forall(c \wedge c_j \wedge \Sigma(G_1) \wedge \Sigma(B_j) \wedge \Sigma(G_2) \rightarrow \Sigma(H))$
and hence Σ is an $\mathcal{A}$-definable model of P_{i+1}.

Obviously, Σ is tight on $Defs_{i+1}$, which is equal to $Defs_i$.

(*Case* R3) Suppose that P_{i+1} is derived from P_i by applying the folding rule. Thus, $P_{i+1} = (P_i \backslash \{C\}) \cup \{E\}$, where C is the clause $H \leftarrow c, G_1, Q, G_2$ in P_i and E is the clause $H \leftarrow e, G_1, K\vartheta, G_2$ derived using the clause D: $K \leftarrow d, B$ in $Defs_i$ according

to rule R3. Moreover, Conditions (i)–(iii) listed above when introducing rule R3, do hold.

Now we show that Σ is an $\mathcal{A}$-definable model of P_{i+1} that is tight on $Defs_{i+1}$. By the hypothesis that Σ is an $\mathcal{A}$-definable model of P_i we have that

$\mathbb{D} \models \forall(c \wedge \Sigma(G_1) \wedge \Sigma(Q) \wedge \Sigma(G_2) \rightarrow \Sigma(H))$.

By Conditions (ii) and (iii) and the definition of symbolic interpretation, we get that

$\mathbb{D} \models \forall(e \wedge \Sigma(G_1) \wedge (\exists Y_1 \ldots \exists Y_m(d \wedge \Sigma(B)))\vartheta \wedge \Sigma(G_2) \rightarrow \Sigma(H))$

where $\{Y_1, \ldots, Y_m\} = Fvars(d \wedge \Sigma(B)) \backslash Fvars(\Sigma(K))$. Since Σ is a symbolic interpretation that is tight on $Defs_i$, we have that

$\mathbb{D} \models \forall(e \wedge \Sigma(G_1) \wedge \Sigma(K\vartheta) \wedge \Sigma(G_2) \rightarrow \Sigma(H))$.

Thus, Σ is an $\mathcal{A}$-definable model of P_{i+1}.

Obviously, Σ is tight on $Defs_{i+1}$, which is equal to $Defs_i$.

(*Case* R4) Suppose that P_{i+1} is derived from P_i by applying the constraint replacement rule. Thus, $P_{i+1} = (P_i \backslash \{(H \leftarrow c_1, G), \ldots, (H \leftarrow c_k, G)\}) \cup \{(H \leftarrow d_1, G), \ldots, (H \leftarrow d_m, G)\}$, where

$\mathbb{D} \models \forall (\exists Y_1 \ldots \exists Y_r \ (c_1 \vee \ldots \vee c_k) \leftrightarrow \exists Z_1 \ldots \exists Z_s \ (d_1 \vee \ldots \vee d_m))$,

$\{Y_1, \ldots, Y_r\} = Fvars(c_1 \vee \ldots \vee c_k) \backslash vars(\{H, G\})$, and $\{Z_1, \ldots, Z_s\} = Fvars(d_1 \vee \ldots \vee d_m) \backslash vars(\{H, G\})$.

Now we show that Σ is an $\mathcal{A}$-definable model of P_{i+1} that is tight on $Defs_{i+1}$. By the hypothesis that Σ is an $\mathcal{A}$-definable model of P_i, the fact that $Y_1, \ldots, Y_r$ do not occur in (G, H), and the distributivity law, we have that

$\mathbb{D} \models \forall(\exists Y_1 \ldots \exists Y_r \ (c_1 \vee \ldots \vee c_k) \wedge \Sigma(G) \rightarrow \Sigma(H))$

and hence

$\mathbb{D} \models \forall(\exists Z_1 \ldots \exists Z_s \ (d_1 \vee \ldots \vee d_m) \wedge \Sigma(G) \rightarrow \Sigma(H))$.

Thus, by using again the distributivity law, and the fact that $Z_1, \ldots, Z_s$ do not occur in (G, H), we get that Σ is an $\mathcal{A}$-definable model of P_{i+1}. Moreover, Σ is tight on $Defs_{i+1}$, which is equal to $Defs_i$. □

Proof of Theorem 5

Proof

Let us assume that there exists an $\mathcal{A}$-definable model Σ' of P_{i+1}, for $i = 0, \ldots, n-1$. We will construct an $\mathcal{A}$-definable model Σ of P_i. The proof proceeds by cases on the transformation rule applied to derive P_{i+1} from P_i.

(*Case* R1) Suppose that P_{i+1} is derived from P_i by applying the definition rule. Thus, $P_{i+1} = P_i \cup \{D\}$, where D is a new clause. Let Σ be equal to Σ'. Σ is an $\mathcal{A}$-definable model of every subset of P_{i+1}, and hence it is an $\mathcal{A}$-definable model of P_i.

(*Case* R2) Suppose that P_{i+1} is derived from P_i by applying the unfolding rule. Thus, $P_{i+1} = (P_i \backslash \{C\}) \cup \{H \leftarrow c, c_j, G_1, B_j, G_2 \mid j = 1, \ldots, m\}$, where C is the clause $H \leftarrow c, G_1, p(X_1, \ldots, X_k), G_2$ in P_i and $\{p(X_1, \ldots, X_k) \leftarrow c_j, B_j \mid j = 1, \ldots, m\}$ is the set of clauses in P_i whose head predicate is p. From the hypothesis that the transformation is not a self-unfolding, that is, H is either *false* or its

predicate is different from p, it follows that the set $\{p(X_1,\ldots,X_k) \leftarrow c_j, B_j \mid j = 1,\ldots,m\}$ is a subset of P_{i+1}. Since Σ' is an $\mathcal{A}$-definable model of P_{i+1}, we have that, for $j=1,\ldots,m$,

$\mathbb{D} \models \forall(c \wedge c_j \wedge \Sigma'(G_1) \wedge \Sigma'(B_j) \wedge \Sigma'(G_2) \rightarrow \Sigma'(H))$ and (1)

$\mathbb{D} \models \forall(c_j \wedge \Sigma'(B_j) \rightarrow \Sigma'(p(X_1,\ldots,X_k)))$ (2)

Now let us define

$\Sigma(q(X_1,\ldots,X_l)) = \Sigma'(q(X_1,\ldots,X_l))$ for q different from p, (3)

$\Sigma(p(X_1,\ldots,X_k)) = \exists Y_1 \ldots \exists Y_n \bigvee_{j=1}^{m}(c_j \wedge \Sigma'(B_j))$ (4)

where $\{Y_1,\ldots,Y_n\} = Fvars(\bigvee_{j=1}^{m}(c_j \wedge \Sigma'(B_j))) \backslash \{X_1,\ldots,X_k\}$.

From (2) and (4) it follows that

$\mathbb{D} \models \forall(\Sigma(p(X_1,\ldots,X_k)) \rightarrow \Sigma'(p(X_1,\ldots,X_k)))$ (5)

and hence, for every conjunction of atoms G,

$\mathbb{D} \models \forall(\Sigma(G) \rightarrow \Sigma'(G))$ (6)

Now we show that Σ is an $\mathcal{A}$-definable model of P_i, that is, Σ is an $\mathcal{A}$-definable model of each clause D in P_i. We consider the following three subcases.

(*Subcase* 1) D is the clause C: $H \leftarrow c, G_1, p(X_1,\ldots,X_k), G_2$ to which the unfolding rule is applied. By definition of Σ, since H is either *false* or its predicate is different from p, by (3), we get

$\Sigma(H) = \Sigma'(H)$.

and hence, by (1), we get

$\mathbb{D} \models \forall(c \wedge \Sigma'(G_1) \wedge \bigvee_{j=1}^{m}(c_j \wedge \Sigma'(B_j)) \wedge \Sigma'(G_2) \rightarrow \Sigma(H))$

By (4), we get

$\mathbb{D} \models \forall(c \wedge \Sigma'(G_1) \wedge \Sigma(p(X_1,\ldots,X_k)) \wedge \Sigma'(G_2) \rightarrow \Sigma(H))$

and, finally, by (6),

$\mathbb{D} \models \forall(c \wedge \Sigma(G_1) \wedge \Sigma(p(X_1,\ldots,X_k)) \wedge \Sigma(G_2) \rightarrow \Sigma(H))$

(*Subcase* 2) D is one of the clauses $p(X_1,\ldots,X_k) \leftarrow c_j, B_j$ used for unfolding $p(X_1,\ldots,X_k)$ in C. From the definition of $\Sigma(p(X_1,\ldots,X_k))$ given by (4), it follows that

$\mathbb{D} \models \forall(c_j \wedge \Sigma'(B_j) \rightarrow \Sigma(p(X_1,\ldots,X_k)))$

and hence, by (6),

$\mathbb{D} \models \forall(c_j \wedge \Sigma(B_j) \rightarrow \Sigma(p(X_1,\ldots,X_k)))$

(*Subcase* 3) D is a clause in $P_i \backslash (\{C\} \cup \{p(X_1,\ldots,X_k) \leftarrow c_j, B_j \mid j = 1,\ldots,m\})$. Let D be a clause of the form $K \leftarrow e, Q$. Since K is either *false* or its predicate is different from p, by (3) it follows that $\Sigma(K) = \Sigma'(K)$. Since D is different from C, it also belongs to P_{i+1}, and by the hypothesis that Σ' is an $\mathcal{A}$-definable model of P_{i+1}, we have that

$\mathbb{D} \models \forall(e \wedge \Sigma'(Q) \rightarrow \Sigma(K))$

Thus, by (6),

$\mathbb{D} \models \forall(e \wedge \Sigma(Q) \rightarrow \Sigma(K))$.

(*Case* R3) Suppose that P_{i+1} is derived from P_i by applying the folding rule. Thus, $P_{i+1} = (P_i \backslash \{C\}) \cup \{E\}$, where C is the clause $H \leftarrow c, G_1, Q, G_2$ in P_i and E is the clause $H \leftarrow e, G_1, K\vartheta, G_2$ derived using the clause D: $K \leftarrow d, B$ in $Defs_i$ according

to rule R3. Moreover, Conditions (i)–(iii) listed above when introducing rule R3, do hold.

Now we show that Σ' is an $\mathcal{A}$-definable model of P_i, and hence we can take Σ to be equal to Σ'. From the hypothesis that the application of folding is reversible it follows that D belongs to P_{i+1}, and since Σ' is an $\mathcal{A}$-definable model of P_{i+1}, we have that

$\mathbb{D} \models \forall(d \wedge \Sigma'(B) \rightarrow \Sigma'(K))$ and

$\mathbb{D} \models \forall(e \wedge \Sigma'(G_1) \wedge \Sigma'(K\vartheta) \wedge \Sigma'(G_2) \rightarrow \Sigma'(H))$

and hence, by the definition of symbolic interpretation,

$\mathbb{D} \models \forall(e \wedge d\vartheta \wedge \Sigma'(G_1) \wedge \Sigma'(B\vartheta) \wedge \Sigma'(G_2) \rightarrow \Sigma'(H))$

By Conditions (i)–(ii) of rule R3, we get

$\mathbb{D} \models \forall(c \wedge \Sigma'(G_1) \wedge \Sigma'(Q) \wedge \Sigma'(G_2) \rightarrow \Sigma'(H))$

and thus Σ' is an $\mathcal{A}$-definable model of P_i.

(*Case* R4) Suppose that P_{i+1} is derived from P_i by applying the constraint replacement rule. Thus, $P_{i+1} = (P_i \backslash \{(H \leftarrow c_1, G), \ldots, (H \leftarrow c_k, G)\}) \cup \{(H \leftarrow d_1, G), \ldots, (H \leftarrow d_m, G)\}$, where

$\mathbb{D} \models \forall (\exists Y_1 \ldots \exists Y_r \ (c_1 \vee \ldots \vee c_k) \leftrightarrow \exists Z_1 \ldots \exists Z_s \ (d_1 \vee \ldots \vee d_m))$,

$\{Y_1, \ldots, Y_r\} = \mathit{Fvars}(c_1 \vee \ldots \vee c_k) \backslash \mathit{vars}(\{H, G\})$, and $\{Z_1, \ldots, Z_s\} = \mathit{Fvars}(d_1 \vee \ldots \vee d_m) \backslash \mathit{vars}(\{H, G\})$.

Now we show that Σ' is an $\mathcal{A}$-definable model of P_i, and hence we can take Σ to be equal to Σ'. By the hypothesis that Σ' is an $\mathcal{A}$-definable model of P_{i+1}, the fact that $Z_1, \ldots, Z_s$ do not occur in (G, H), and the distributivity law, we have that

$\mathbb{D} \models \forall(\exists Z_1 \ldots \exists Z_s \ (d_1 \vee \ldots \vee d_m) \wedge \Sigma'(G) \rightarrow \Sigma'(H))$

and hence

$\mathbb{D} \models \forall(\exists Y_1 \ldots \exists Y_r \ (c_1 \vee \ldots \vee c_k) \wedge \Sigma'(G) \rightarrow \Sigma'(H))$.

Thus, by using again the distributivity law and the fact that $Y_1, \ldots, Y_r$ do not occur in (G, H), we get that Σ' is an $\mathcal{A}$-definable model of P_i. □