Verifying relational program properties by transforming constrained Horn clauses

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

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

Abstract. We present a method for verifying relational program properties, that is, properties that relate the input and the output of two programs. Our verification method is parametric with respect to the definition of the semantics of the programming language in which the programs are written. That definition consists of a set *Int* of constrained Horn clauses (CHC) that encode the interpreter of the programming language. Then, given the programs and the relational property we want to verify, we generate, by using *Int*, a set of constrained Horn clauses whose satisfiability is equivalent to the validity of the property. Unfortunately, state-of-the-art solvers for CHC have severe limitations in proving the satisfiability, or the unsatisfiability, of such sets of clauses. We propose some transformation techniques that increase the power of CHC solvers when verifying relational properties. We show that these transformations, based on unfolding and folding rules, preserve satisfiability. Through an experimental evaluation we also show that in many cases CHC solvers are able to prove the (un)satisfiability of sets of clauses obtained by applying the transformations we propose, whereas the same solvers are unable to perform those proofs when given as input the original sets of constrained Horn clauses.

1 Introduction

During the process of software development it is often the case that several versions of the same program are produced. This is due to the fact that the programmer, for instance, may want to replace an old program fragment by a new one with the objective of improving efficiency, or adding a new feature, or modifying the program structure. In these cases, in order to prove the correctness of the whole program, it may be desirable to consider *relational properties* of those program fragments, that is, properties that relate the semantics of the old fragments to the semantics of the new fragments. A particular example of a relational property is *program equivalence*, but many other relations may be considered (see [5,29] for other significant examples).

It has been noted that proving relational properties between two structurally similar program versions is often easier than directly proving the desired correctness properties for the new program version [5,21,29]. Moreover, in order to automate the proof of such properties, it may be convenient to follow a transformational approach so that one can use the already available methods and tools for proving correctness properties of individual programs. For instance, in [5,29,35] the authors propose program composition and cross-product techniques such that, in order to prove that a given relation between program P_1 and P_2 holds, it is sufficient to show that suitable pre- and post-conditions for the composition of P_1 and P_2 hold. The validity of these pre- and postconditions is then checked by using state-of-the-art verification tools (e.g., BOOGIE [4] and WHY [22]). A different transformational approach is followed by [21]. The authors of [21] introduce a set of proof rules for program equivalence, and from these rules they generate *verification conditions* in the form of *constrained Horn clauses* (CHCs) which is a logical formalism recently suggested for program verification (see [7] for a survey of the techniques which use CHCs). The satisfiability of the verification conditions, which guarantees that the relational property holds, can be checked by using CHC solvers, such as ELDARICA [25], MATHSAT [11], and Z3 [18] (obviously, no complete solver exists because most properties of interest, including equivalence, are in general undecidable).

Unfortunately, all the above mentioned approaches enable only a partial reuse of the available verification techniques, because one has to develop specific transformation rules for each programming language and each proof system in use.

In this paper we propose a method to achieve a higher parametricity with respect to the programming language and the class of relational properties considered, and this is done by pushing further the transformational approach.

As a first step of our method, we formalize a relational property between two programs as a set of CHCs. This is done by using an interpreter for the given programming language written in clausal form as indicated in [17,31]. In particular, the properties of the data domain in use, such as the integers and the integer arrays, are formalized in the constraint theory of the CHCs.

Now, it is very often the case that this first step is not sufficient to allow state-of-the-art CHC solvers to verify the properties of interest. Indeed, the strategies for checking satisfiability employed by those solvers deal with the sets of clauses encoding the semantics of each of the two programs in an independent way, thereby failing to take full advantage of the interrelations between the two sets of clauses. In this paper, instead of looking for a new, smarter strategy for satisfiability checking, we propose some transformation techniques for CHCs that compose together, in a suitable way, the clauses relative to the two programs, so that their interrelations may be better exploited. This transformational approach has the advantage that we can use existing techniques for CHC satisfiability as a final step of the verification process. Moreover, since the CHC encodings of the two programs do not explicitly refer to the syntax of the given programs, we are able to prove relations that would be difficult to infer by the above men-

tioned, syntax-driven approaches. Our approach has been proved to be effective in practice, as indicated in Section 5.

The main contributions of the paper are the following.

(1) We present a method for encoding as a set of CHCs a large class of relational properties of programs written in a simple imperative language. The only language-specific element of our method is that we need a CHC interpreter that provides a formal definition of the semantics of the programming language in use.

(2) We propose an automatic transformation technique for CHCs, called *predicate pairing*, that combines together the clauses representing the semantics of each program with the objective of increasing the effectiveness of the subsequent application of the CHC solver at hand. We prove that predicate pairing guarantees equisatisfiability between the initial and the final sets of clauses. The proof is based on the fact that this transformation can be expressed as a sequence of applications of the *unfolding* and *folding* rules [20,33].

(3) We report on an experimental evaluation performed by a proof-of-concept implementation of our transformation technique by using the VeriMAP system [15]. The satisfiability of the transformed CHC is then verified by using the solvers Eldarica [25], MathSAT [11], and Z3 [18]. Our experiments show that the transformation is effective on a number of small, but nontrivial examples. Moreover, our method is competitive with respect to some special purpose tools for checking equivalence [21].

The paper is organized as follows. We start off by presenting in Section 2 a simple introductory example. Then, in Section 3 we present the translation of a relational property between two programs into constrained Horn clauses. In Section 4 we present the various transformation techniques and we prove that they preserve satisfiability (and unsatisfiability). The implementation of the verification method and its experimental evaluation is reported in Section 5. Finally, in Section 6, we discuss the related work.

2 An Introductory Example

In this section we present an example to illustrate the main ideas developed in this paper. Let us consider the two programs of Fig. 1.
Program `sum_upto` computes the sum of the first `x1` non-negative integer numbers and program `prod` computes the product of `x2` by `y2` by summing up `x2` times the value of `y2`. Thus, we have the following property *Leq*: if $\texttt{x1} = \texttt{x2}$ and $\texttt{x2} \leq \texttt{y2}$, then the two programs return the values `z1` and `z2`, respectively, such that $\texttt{z1} \leq \texttt{z2}$. By using the method we will present in Section 3 (and the CHC specialization of Section 4.1), the property *Leq* relating the computations of the two programs is translated into the set of constrained Horn clauses listed in Fig. 2, written in the syntax of Constraint Logic Programming (CLP) [26].
Clause 1 specifies the relational property *Leq*, where primed logical variables refer to the final values of the corresponding imperative variables. In particular,

```
void sum_upto() {
  z1=f(x1);
}
int f(int n1){
  int r1;
  if (n1 <= 0) {
    r1 = 0;
  } else {
    r1 = f(n1 - 1) + n1;
  }
  return r1;
}
```

```
void prod() {
  z2 = g(x2,y2);
}
int g(int n2, int m2){
  int r2;
  r2=0;
  while (n2 > 0) {
    r2 += m2;
    n2--;
  }
  return r2;
}
```

Relational property *Leq*: $\{\texttt{x1}=\texttt{x2},\ \texttt{x2}\leq\texttt{y2}\}\ \texttt{sum\_upto} \sim \texttt{prod}\ \{\texttt{z1}\leq\texttt{z2}\}$

Fig. 1. The programs `sum_upto` and `prod`, and the relational property to be proved.

1. $\mathit{false} \leftarrow X1 = X2, X2 \leq Y2, Z1' > Z2', su(X1, Z1'), p(X2, Y2, Z2')$
2. $su(X1, Z1') \leftarrow f(X1, Z, X1, R, N1, Z1')$
3. $f(X, Z, N, R, N, 0) \leftarrow N \leq 0$
4. $f(X, Z, N, R, N, Z1) \leftarrow N \geq 1, N1 = N-1, Z1 = R2+N, f(X, Z, N1, R1, N2, R2)$
5. $p(X2, Y2, Z2') \leftarrow g(X2, Y2, Z, X2, Y2, 0, N, P, Z2')$
6. $g(X, Y, Z, N, P, R, N, P, R) \leftarrow N \leq 0$
7. $g(X, Y, Z, N, P, R, N2, P2, R2) \leftarrow N \geq 1, N1 = N-1, R1 = P+R,$
   $g(X, Y, Z, N1, P, R1, N2, P2, R2)$

Fig. 2. Translation into constrained Horn clauses of the relational property *Leq*.

note that the constraint $Z1' > Z2'$ is the negation of the property we want to prove. Clauses 2–4 and 5–7 encode the input-output relations computed by programs `sum_upto` and `prod`, respectively. The relational property *Leq* holds iff clauses 1–7 are satisfiable.

Unfortunately, state-of-the-art solvers for constrained Horn clauses with linear integer arithmetic (such as Eldarica [25], MathSAT [11], and Z3 [18]) are unable to prove the satisfiability of clauses 1–7. This is due to the fact that those solvers reason on the predicate *su* and *p* separately, and hence, to prove that clause 1 is satisfiable (that is, its premise is unsatisfiable), they should discover quadratic relations among variables (in our case, $Z1' = X1 \times (X1-1)/2$ and $Z2' = X2 \times Y2$), and these relations cannot be expressed by linear arithmetic constraints.

In order to deal with this limitation one could extend constrained Horn clauses with solvers for the theory of non-linear integer arithmetic constraints [8]. However, this extension has to cope with the additional problem that the satisfiability problem for non-linear constraints is undecidable [30].

In this paper we propose an approach based on suitable transformations of the clauses that encode the property *Leq* into an equisatisfiable set of clauses, which, as shown in the sequel, are hopefully easier to solve. In our example, the clauses of Fig. 2 are transformed into the ones shown in Fig. 3.

The predicate $fg(X, Z, N, R, N1, Z1, Y2, V, W, N2, P2, Z2)$ is equivalent to the conjunction $f(X, Z, N, R, N1, Z1), g(X, Y2, V, N, Y2, W, N2, P2, Z2)$. The ef-

$$
\begin{array}{l}
\mathit{false} \leftarrow X1 \leq Y2,\ Z1' > Z2',\ fg(X1,Z,X1,R,N1,Z1',Y2,Z,0,N2,P2,Z2') \\
fg(X,Z,N,R,N,0,Y2,V,Z2,N,P2,Z2) \leftarrow N \leq 0 \\
fg(X,Z,N,R,N,Z1,Y2,V,W,N2,P2,Z2) \leftarrow \\
\qquad\qquad N \geq 1,\ N1 = N-1,\ Z1 = R2+N,\ M = Y2+W, \\
\qquad\qquad fg(X,Z,N1,R1,S,R2,Y2,V,M,N2,P2,Z2)
\end{array}
$$

Fig. 3. Transformed clauses derived from the clauses 1–7 in Fig. 2.

fect of this transformation is that it is possible to infer linear relations among a subset of the variables occurring in the *conjunctions* of predicates, without having to use in an explicit way their non-linear relations with other variables. In particular, one can infer that $fg(X1, Z, X1, R, N1, Z1', Y2, Z, 0, N2, P2, Z2')$ enforces the constraint $(X1 > Y2) \vee (Z1' \leq Z2')$, and hence the satisfiability of the first clause of Fig. 3, without having to derive quadratic relations. Indeed, after this transformation MATHSAT (and, after further transformation, also ELDARICA and Z3) is able to prove the satisfiability of the clauses of Fig. 3, which implies the validity of the relational property *Leq*.

3 Specifying Relational Properties in CHC

In this section we introduce the notion of a relational property relative to two programs written in a simple imperative language and we show how a relational property can be translated into constrained Horn clauses.

3.1 Relational properties

We consider a C-like programming language manipulating integers and integer arrays via assignments, function calls, conditionals, while loops, and goto's. A program is a sequence of labeled commands (or commands, for short), and in each program there is a unique `halt` command that, when executed, causes program termination.

The semantics of our language is defined by a binary *transition relation*, denoted by $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle \ell : c, \delta \rangle\!\rangle$ of a labeled command $\ell : c$ and an *environment* δ. An environment δ is a function that maps every variable identifier x of a set $\mathcal{V}$ of identifiers to its value v in the integers (for integer variables) or in the set of the finite sequences of integers (for array variables). Given an environment δ, $dom(\delta)$ denotes its domain. The definition of the relation $\Longrightarrow$ corresponds to the *multistep* operational semantics, that is: (i) the semantics of each command, other than a function call, is defined by a pair of the form $\langle\!\langle \ell : c, \delta \rangle\!\rangle \Longrightarrow \langle\!\langle \ell' : c', \delta' \rangle\!\rangle$, and (ii) the semantics of a function call is recursively defined in terms of the reflexive, transitive closure $\Longrightarrow^*$.

In particular, the semantics of an assignment is:

$$(R1) \qquad \langle\!\langle \ell : x = e,\ \delta \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)),\ update(\delta, x, [\![e]\!]\,\delta) \rangle\!\rangle$$

where: (i) $at(\ell)$ denotes the command whose label is ℓ, (ii) $nextlab(\ell)$ denotes the label of the command which is *immediately after* the command with label ℓ, (iii)

$update(\delta, x, v)$ denotes the environment δ' that is equal to the environment δ, except that $\delta'(x) = v$, and (iv) $[\![e]\!]\,\delta$ is the value of the expression e in the environment δ.

The semantics of a call to the function f is:

$$(R2) \qquad \langle\!\langle \ell : x = f(e_1, \ldots, e_k),\ \delta \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)), update(\delta', x, [\![e]\!]\,\delta') \rangle\!\rangle$$
$$\text{if } \langle\!\langle at(firstlab(f)),\ \overline{\delta} \rangle\!\rangle \implies^* \langle\!\langle \ell_r : \texttt{return}\ e,\ \delta' \rangle\!\rangle$$

where: (i) $firstlab(f)$ denotes the label of the first command in the definition of the function f, and (ii) $\overline{\delta}$ is the environment δ extended by the bindings for the formal parameters, say $x_1, \ldots, x_k$, and the local variables, say $y_1, \ldots, y_h$, of f (we assume that the identifiers x_i's and y_i's do not occur in $dom(\delta)$). Thus, we have that $\overline{\delta} = \delta \cup \{x_1 \mapsto [\![e_1]\!]\,\delta, \ldots, x_k \mapsto [\![e_k]\!]\,\delta, y_1 \mapsto v_1, \ldots, y_h \mapsto v_h\}$, for arbitrary values $v_1, \ldots, v_h$. We refer to [17] for a more detailed presentation of the multistep semantics.

A program P *terminates* for an initial environment δ and computes the final environment η, denoted $\langle P, \delta \rangle \Downarrow \eta$, iff $\langle\!\langle \ell_0 : c, \delta \rangle\!\rangle \implies^* \langle\!\langle \ell_h : \texttt{halt}, \eta \rangle\!\rangle$, where $\ell_0 : c$ is the first labeled command of P. $\langle\!\langle \ell_0 : c, \delta \rangle\!\rangle$ and $\langle\!\langle \ell_h : \texttt{halt}, \eta \rangle\!\rangle$ are called the *initial configuration* and the *final configuration*, respectively.

Now, we can formally define a relational property as follows. Let P_1 and P_2 be two programs with global variables in $\mathcal{V}_1$ and $\mathcal{V}_2$, respectively, with $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$. Let φ and ψ be two first order formulas with variables in $\mathcal{V}_1 \cup \mathcal{V}_2$. Then, by using the notation of [5], a relational property is specified by the 4-tuple $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$. For instance, the relational property *Leq* presented in Section 2 is specified by: $\{\texttt{x1} = \texttt{x2},\ \texttt{x2} \leq \texttt{y2}\}\ \texttt{sum\_upto} \sim \texttt{prod}\ \{\texttt{z1} \leq \texttt{z2}\}$.

We say that $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$ is valid iff the following holds: if the inputs of P_1 and P_2 satisfy the *pre-relation* φ and the programs P_1 and P_2 both terminate, then upon termination the outputs of P_1 and P_2 satisfy the *post-relation* ψ. The validity of a relational property is formalized by Definition 1 below, where given a formula α and an environment δ, by $\alpha\,[\delta]$ we denote the formula α where every free occurrence of a variable has been replaced by its values in δ.

Definition 1. *A relational property* $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$ *is said to be* valid*, denoted* $\models \{\varphi\}\ P_1 \sim P_2\ \{\psi\}$*, iff for all environments* δ_1 *and* δ_2 *with* $dom(\delta_1) \subseteq \mathcal{V}_1$ *and* $dom(\delta_2) \subseteq \mathcal{V}_2$*, the following holds*:

if $\models \varphi\,[\delta_1 \cup \delta_2]$ *and* $\langle P_1, \delta_1 \rangle \Downarrow \eta_1$ *and* $\langle P_2, \delta_2 \rangle \Downarrow \eta_2$*, then* $\models \psi\,[\eta_1 \cup \eta_2]$.

3.2 Formal Semantics of the Imperative Language in CHC

In order to translate a relational program property into constrained Horn clauses, first we need to specify the operational semantics of our C-like language by a set of constrained Horn clauses. We follow the approach presented in [17] which now we briefly recall.

The transition relation $\implies$ between configurations and its reflexive, transitive closure $\implies^*$ are specified by the binary predicates *tr* and *reach*, respectively. We only show the formalization of the semantic rules $R1$ and $R2$ above, consisting of the following clauses $D1$ and $D2$, respectively. For the other rules of the multistep operational semantics we refer to [17].

$(D1)$ $tr(cf(cmd(L, asgn(X, expr(E))), Env), cf(cmd(L1, C), Env1)) \leftarrow$
$eval(E, Env, V), update(Env, X, V, Env1), nextlab(L, L1), at(L1, C)$

$(D2)$ $tr(cf(cmd(L, asgn(X, call(F, Es))), Env), cf(cmd(L2, C2), Env2)) \leftarrow$
$fun\_env(Es, Env, F, FEnv), firstlab(F, FL), at(FL, C),$
$reach(cf(cmd(FL, C), FEnv), cf(cmd(LR, return(E)), Env1)),$
$eval(E, Env1, V), update(Env1, X, V, Env2), nextlab(L, L2), at(L2, C2)$

The predicate *reach* is recursively defined by the following two clauses:

$reach(C, C)$
$reach(C, C2) \leftarrow tr(C, C1), reach(C1, C2)$

Configurations are represented by using terms of the form $cf(cmd(L, C), Env)$, where: (i) L and C encode the label and the command, respectively, (ii) *Env* encodes the environment. The term $asgn(X, expr(E))$ encodes the assignment of the value of the expression E to the variable X. The predicate $eval(E, Env, V)$ holds iff V is the value of the expression E in the environment *Env*. The term $call(F, Es)$ encodes the call of the function F with the list *Es* of the actual parameters. The predicate $fun\_env(Es, Env, F, FEnv)$ computes from *Es* and *Env* the list *Vs* of the values of the actual parameters of the function F and builds the new initial environment *FEnv* for executing the body of F. In *FEnv* the local variables of F are all bound to arbitrary values. The other terms and predicates occurring in clauses $D1$ and $D2$ have the obvious meaning which can be derived from the above explanation of the semantic rules $R1$ and $R2$.

Given a program *Prog*, represented as a set of $at(L, C)$ facts, its input/output relation is represented by a predicate *prog* defined as follows:

$prog(X, X') \leftarrow initConf(C, X), reach(C, C'), finalConf(C', X')$

where $initConf(C, X)$ and $finalConf(C', X')$ hold iff the tuples X and X' are the values of the global variables of *Prog* in the initial and final configurations C and C', respectively.

3.3 Translating Relational Properties into CHC

Let us consider a relational property $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$. We assume that φ and ψ are quantifier-free formulas of the theory $\mathcal{A}$ of linear integer arithmetic and integer arrays [9]. A quantifier free formula of $\mathcal{A}$ is also called a *constraint*.

More complex theories of constraints may be used for defining relational properties. For instance, one may consider theories with nested quantifiers [2]. Our approach is, to a large extent, parametric with respect to those theories. Indeed, the transformation rules on which it is based only require that satisfiability and entailment of constraints be decidable (see Section 4).

The validity of a relational property is translated into a set of constrained Horn clauses, that is, implications of the form: $A_0 \leftarrow c, A_1, \ldots, A_n$, where (i) A_0 is either an atomic formula (or *atom*) or *false*, (ii) c is a constraint, and (iii) $A_1, \ldots, A_n$ is a possibly empty conjunction of atoms. A set S of clauses is *$\mathcal{A}$-satisfiable* or, simply, *satisfiable* iff $\mathcal{A} \cup S$ is satisfiable.

The relational property of the form $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$ is translated into the following clause:

$$(Prop) \qquad \mathit{false} \leftarrow \mathit{pre}(X,Y), p1(X,X'), p2(Y,Y'), \mathit{neg\_post}(X',Y')$$

where: (i) X and Y are the disjoint tuples of global variables of P_1 and P_2, respectively (in the translation we use capital letters for variable identifiers);
(ii) X' and Y' are primed versions of X and Y, respectively;
(iii) $\mathit{pre}(X,Y)$ is φ;
(iv) the predicates $p1(X,X')$ and $p2(Y,Y')$ are defined by a set of clauses derived from P_1 and P_2, respectively, by using a formalization of the the operational semantics of the programming language, as explained in Section 3.2; and
(v) $\mathit{neg\_post}(X',Y')$ is $\neg\psi$, with all variables replaced by their primed versions.

Note that we can always eliminate negation from the atoms $\mathit{pre}(X,Y)$ and $\mathit{neg\_post}(X',Y')$ by pushing negation inward and transforming negated equalities into disjunctions of inequalities. Moreover, we can eliminate disjunction from constraints and replace clause *Prop* by a set of clauses with *false* in the head. Although these transformations are not strictly needed by the techniques described in the rest of the paper, they may be helpful for the constraint solving tools we use when automating our verification method.

Let RP be a relational property and T_{RP} be the set of constrained Horn clauses generated by the translation process described above, then T_{RP} is correct in the following sense.

Theorem 1 (Correctness of the CHC Translation). *RP is valid iff T_{RP} is satisfiable.*

The proof of this theorem directly follows from the fact that the predicate *reach* is a correct formalization of the semantics of the programming language.

4 Transforming Specifications of Relational Properties

The reduction of the validity problem, that is, the problem of testing whether or not $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$ is valid, to the problem of verifying the satisfiability of a set T_{RP} of constrained Horn clauses allows us to apply reasoning techniques that are independent of the specific programming language in which programs P_1 and P_2 are written. In particular, we can try to solve the satisfiability problem for T_{RP} by applying one of the available solvers for constrained Horn clauses. Unfortunately, as shown by the example in Section 2, it may be the case that these solvers fail to prove satisfiability (or unsatisfiability). In Section 5 the reader will find an experimental evidence of this limitation. However, a very significant advantage of having reduced the validity problem to a CHC satisfiability problem is that we can now transform the set T_{RP} by applying any satisfiability preserving algorithm, before submitting the new, transformed satisfiability problem to a CHC solver.

In this section we present some transformations of constrained Horn clauses that have the objective of increasing the effectiveness of the subsequent uses

of CHC solvers. These transformations, called *transformation strategies*, are: (1) *CHC specialization*, and (2) *predicate pairing*.

These strategies are variants of techniques developed in the area of logic programming for improving the efficiency of program execution [19,32]. We will show that these techniques are very effective for the class of verification problems we are considering here.

The above CHC Specialization and Predicate Pairing strategies are realized as sequences of applications of some elementary *transformation rules*, collectively called *unfold/fold rules*, proposed in the field of CLP [20].

Now we present the version of those rules we need in our context here. Those rules allow us to derive from an old set *Cls* of constrained Horn clauses a new set *TransfCls* of constrained Horn clauses.

Definition Rule. We introduce a *definition clause* D of the form $newp(X) \leftarrow c, G$, where *newp* is a new predicate symbol, X is a tuple of variables occurring in $\{c, G\}$, c is a constraint, and G is a non-empty conjunction of atoms. We derive the set of clauses $\mathit{TransfCls} = \mathit{Cls} \cup \{D\}$. We denote by *Defs* the set of all definition clauses introduced in a sequence of application of the unfold/fold rules.

Unfolding Rule. Given a clause C in *Cls* of the form $H \leftarrow c, L, A, R$, where H is either *false* or an atom, A is an atom, c is a constraint, and L and R are (possibly empty) conjunctions of atoms, let us consider the set $\{K_i \leftarrow c_i, B_i \mid i = 1, \ldots, m\}$ made out of the (renamed apart) clauses of *Cls* such that, for $i = 1, \ldots, m$, A is unifiable with K_i via the most general unifier ϑ_i and $\mathcal{A} \models \exists X.(c, c_i)\,\vartheta_i$ where X is the tuple of variables in $(c, c_i)\,\vartheta_i$. By unfolding C w.r.t. A using *Cls*, we derive the set of clauses $\mathit{TransfCls} = (\mathit{Cls} - \{C\}) \cup U(C)$, where the set $U(C)$ is $\{(H \leftarrow c, c_i, L, B_i, R)\,\vartheta_i \mid i = 1, \ldots, m\}$.

Folding Rule. Given a clause E of the form: $H \leftarrow e, L, Q, R$ and a clause D in *Defs* of the form $K \leftarrow d, G$ such that: (i) for some substitution ϑ, $Q = G\,\vartheta$, and (ii) $\mathcal{A} \models \forall X.\,(e \rightarrow d\,\vartheta)$ holds, where X is the tuple of variables in $e \rightarrow d\,\vartheta$, then by folding E using D we derive the set of clauses $\mathit{TransfCls} = (\mathit{Cls} - \{E\}) \cup \{H \leftarrow e, L, K\vartheta, R\}$.

By using the results in [20], which ensure that the transformation rules preserve the least model of a set of constrained Horn clauses, if any, we get the following result.

Theorem 2 (Soundness of the Unfold/Fold Rules). *Suppose that from a set Cls of constrained Horn clauses we derive a new set TransfCls of clauses by a sequence of applications of the* unfold/fold rules*, where every definition clause used for folding is unfolded during that sequence. Then Cls is satisfiable iff TransfCls is satisfiable.*

4.1 CHC Specialization

Specialization is a transformation technique that has been proposed in various programming contexts to take advantage of static information to simplify and

customize programs [27]. In the field of program verification it has been shown that the specialization of constrained Horn clauses can be very useful to simplify clauses before checking their satisfiability [13,28].

We will use CHC specialization to simplify our initial set of clauses T_{RP}. In particular, starting from clause *Prop* of Section 3.3, we introduce two new predicates $p1_{sp}$ and $p2_{sp}$, defined by the clauses:

$(S1)\ \ p1_{sp}(V, V') \leftarrow p1(X, X') \qquad\qquad (S2)\ \ p2_{sp}(W, W') \leftarrow p2(Y, Y')$

where V, V', W, W' are the sub-tuples of X, X', Y, Y', respectively, which occur in $pre(X, Y)$ or $neg\_post(X', Y')$. Then, by applying the folding rule, we can replace $p1$ and $p2$ in clause *Prop*, by $p1_{sp}$ and $p2_{sp}$, thereby obtaining:

$(Prop_{sp})\qquad false \leftarrow pre(V, W), p1_{sp}(V, V'), p2_{sp}(W, W'), neg\_post(W, W')$

Now, by applying the specialization strategy of [13] starting from the set of clauses $(T_{RP} - \{Prop\}) \cup \{Prop_{sp}, S1, S2\}$, we derive specialized versions of the clauses that define the semantics of programs P_1 and P_2. In particular, in those clauses there are reference to neither the predicate *reach*, nor the predicate *tr*, nor the terms encoding configurations.

For instance, let us consider again our example of Section 2. The translation of the relational property *Leq*: $\{\texttt{x1} = \texttt{x2},\ \texttt{x2} \leq \texttt{y2}\}\ \texttt{sum\_upto} \sim \texttt{prod}\ \{\texttt{z1} \leq \texttt{z2}\}$ is as follows:

$$false \leftarrow X1 = X2, X2 \leq Y2, Z1' > Z2',$$
$$sum\_upto(X1, Z1, X1', Z1'),\ prod(X2, Y2, Z2, X2', Y2', Z2')$$

where predicates *sum_upto* and *prod* are defined in terms of the predicate *reach* as shown in Section 3.2. By specialization we get clauses 1–7 of Fig. 2, where *su* and *p* are the specialized versions of *sum_upto* and *prod*, respectively.

CHC specialization is performed by applying the unfold/fold rules, and hence by Theorem 2 the following property holds.

Theorem 3. *Let T_{sp} be derived from T_{RP} by specialization. Then T_{RP} is satisfiable iff T_{sp} is satisfiable.*

4.2 Predicate Pairing

The core of our verification method is the predicate pairing transformation strategy (see Fig. 4), which composes pairs of predicates q and r into one new predicate t equivalent to their conjunction. As suggested by the example of Section 2, this transformation may ease the discovery of relations between variables occurring in the two original predicates, and thus it may ease the satisfiability test.

Let us see the predicate pairing strategy in action by considering again the example of Section 2.

First Iteration of the while loop.

UNFOLDING: The strategy starts off by unfolding $su(X1, Z1')$ and $p(X2, Y2, Z2')$ in clause 1 of Fig. 2, hence deriving the following new clause (in which we have also replaced $X2$ by $X1$, by applying the equality $X1 = X2$):

Input: A set $Q \cup R \cup \{C\}$ of clauses where: (i) C is of the form $\mathit{false} \leftarrow c, q(X), r(Y)$, (ii) q and r occur in Q and R, respectively, and (iii) no predicate occurs in both Q and R.
Output: A set *TransfCls* of clauses.

INITIALIZATION: $\mathit{InCls} := \{C\}$; $\mathit{Defs} := \emptyset$; $\mathit{TransfCls} := Q \cup R$;

while there is a clause C in *InCls* *do*
  UNFOLDING: From clause C derive a set $U(C)$ of clauses by unfolding C with respect to every atom occurring in its body using $Q \cup R$;
  Rewrite each clause in $U(C)$ to a clause of the form $H \leftarrow d, A_1, \ldots, A_k$, such that, for $i = 1, \ldots, k$, A_i is of the form $p(X_1, \ldots, X_m)$, where $X_1, \ldots, X_m$ are, not necessarily distinct, variables;
  DEFINITION & FOLDING:
  $F(C) := U(C)$;
  for every clause $E \in F(C)$ of the form $H \leftarrow d, G_1, q_i(V_i), G_2, r_i(W_i), G_3$ where q_i and r_i occur in Q and R, respectively, *do*
    if in *Defs* there is no clause of the form $\mathit{newp}(Z) \leftarrow q_i(V_i), r_i(W_i)$, where Z is the tuple of distinct variables in (V_i, W_i)
    then add $\mathit{newp}(Z) \leftarrow q_i(V_i), r_i(W_i)$ to *InCls* and to *Defs*;
    $F(C) := (F(C) - \{E\}) \cup \{H \leftarrow d, G_1, \mathit{newp}(Z), G_2, G_3\}$
  end-for
  $\mathit{InCls} := \mathit{InCls} - \{C\}$; $\mathit{TransfCls} := \mathit{TransfCls} \cup F(C)$;
end-while

Fig. 4. The *predicate pairing* transformation strategy.

8. $\mathit{false} \leftarrow X1 \leq Y2, Z1' > Z2',$
   $f(X1, Z, X1, R, N1, Z1'), g(X1, Y2, Z, X1, Y2, 0, N2, P2, Z2')$

DEFINITION & FOLDING: A new atom with predicate *fg* is introduced for replacing the conjunction of the atoms with predicates f and g in the premise of clause 8:

9. $fg(X1, Z, N, R, N1, Z1', Y2, V, W, N2, P2, Z2') \leftarrow$
   $f(X1, Z, N, R, N1, Z1'), g(X1, Y2, V, N, Y2, W, N2, P2, Z2')$

and that conjunction is folded, hence deriving:

10. $\mathit{false} \leftarrow X1 \leq Y2, Z1' > Z2', fg(X1, Z, X1, R, N1, Z1', Y2, Z, 0, N2, P2, Z2')$

Second Iteration of the while loop.

UNFOLDING: Now, the atoms with predicate f and g in the premise of the newly introduced clause 9 are unfolded, and the following new clauses are derived:

11. $fg(X, Z, N, R, N, 0, Y, V, W, N2, Y, Z2) \leftarrow N \leq 0$
12. $fg(X, Z, N, R, N, Z1, Y2, V, W, N2, P2, Z2) \leftarrow$
    $N \geq 1, N1 = N - 1, Z1 = R2 + N, M = Y2 + W,$
    $f(X, Z, N1, R1, S, R2), g(X, Y2, V, N1, Y2, M, N2, P2, Z2)$

DEFINITION & FOLDING: No new predicate is needed, as the conjunction of the atoms with predicate f and g in clause 12 can be folded using clause 9. We get:

13. $fg(X, Z, N, R, N, Z1, Y2, V, W, N2, P2, Z2) \leftarrow$
$N \geq 1, N1 = N - 1, Z1 = R2 + N, M = Y2 + W,$
$fg(X, Z, N1, R1, S, R2, Y2, V, M, N2, P2, Z2)$

Clauses 10, 11, and 13, which are the ones shown in Fig. 3, constitute the final set of clauses we have derived.

The predicate pairing strategy always terminates because the number of the possible new predicate definitions is bounded by the number γ of conjunctions of the form $q_i(V_i), r_i(W_i)$, where q_i occurs in Q and r_i occurs in R and, hence, the number of executions of the *while* loop of the strategy is bounded by γ.

Thus, from the fact that the unfold/fold transformation rules preserve satisfiability (see Theorem 2), we get the following result.

Theorem 4 (Soundness of the predicate pairing strategy). *Let the set $Q \cup R \cup \{C\}$ of clauses be the input of the predicate pairing strategy. Then the strategy terminates and returns a set TransfCls of clauses such that $Q \cup R \cup \{C\}$ is satisfiable iff TransfCls is satisfiable.*

5 Implementation and Experimental Evaluation

We have implemented the techniques presented in Sections 3 and 4 by using the VeriMAP transformation and verification system [15]. The satisfiability of the constrained Horn clauses derived by transformation has been checked by using the SMT solvers ELDARICA [25], MATHSAT [11], and Z3 [18].

We have considered 90 problems[4]. referring to relational properties of small, yet non-trivial, C programs mostly taken from the literature [5,6,21]. The properties we have considered belong to the following categories. All programs act on integers, except for those in the ARR category which act on integer arrays. The ITE (respectively, REC) category consists of equivalence properties between pairs of iterative (respectively, recursive) programs, that is, we have verified that for every pair of programs, the two programs in the pair compute the same outputs when given the same inputs. The I-R category consists of equivalence properties between an iterative and a recursive (non-tail recursive) program. For example, we have verified the equivalence of iterative and recursive versions of programs computing the greatest common divisor of two integers and the n-th triangular number $T_n = \sum_{i=1}^{n} i$. The ARR category consists of equivalence properties between programs acting on integer arrays. The LEQ category consists of inequality properties stating that if the inputs of two programs satisfy some given precondition, then their outputs satisfy an inequality postcondition. For instance, we have verified that for all non-negative integers m and n: (i) if $n \leq m$, then $T_n \leq n \times m$ (see the example of Section 2), and (ii) $n^2 \leq n^3$. The MON (respectively, INJ) category consists of properties stating that programs, under some given preconditions, compute monotonically non-decreasing (respectively, injective) functions. For example, we have verified monotonicity and injectivity

[4] The C sources are available at `http://map.uniroma2.it/cilc16/sources.tar.gz`

Cat	*n*	*Tr*	*Eld*	*Z3*	*PP*	*Eld*	*MS*	*Z3*	*CP+Eld*	*CP+MS*	*CP+Z3*
ITE	21	0.04	7 (5.61)	6 (1.06)	1.23	13 (5.10)	19 (7.39)	6 (1.08)	4 (0.07)	1 (0.10)	15 (2.55)
REC	18	0.06	7 (4.16)	8 (3.10)	1.82	7 (5.19)	11 (1.09)	6 (0.94)	7 (0.06)	0	7 (0.08)
I-R	4	0.05	0	0	0.58	0	3 (8.16)	0	4 (0.35)	1 (0.08)	4 (0.35)
ARR	5	0.05	0	1 (0.78)	1.02	1 (4.29)	1 (0.78)	4 (0.81)	1 (5.81)	1 (2.26)	1 (0.75)
LEQ	6	0.03	1 (2.83)	1 (0.77)	0.29	1 (4.76)	6 (2.52)	1 (0.80)	2 (1.74)	0	3 (0.64)
MON	18	0.02	4 (4.60)	4 (0.90)	2.78	11 (5.55)	16 (1.61)	8 (0.98)	0	1 (3.12)	6 (0.81)
INJ	11	0.02	0	0	1.87	0	11 (1.70)	4 (1.10)	7 (1.71)	0	6 (0.63)
FUN	7	0.02	5 (4.49)	5 (0.80)	3.90	6 (4.77)	7 (1.03)	5 (0.91)	0	0	2 (0.69)
Tot	90	0.04	24(4.67)	25(1.61)	1.84	39(5.16)	74(3.30)	34(0.97)	25(0.93)	4(1.39)	44(1.20)

Table 1. Timings are in seconds. The timeout occurs after 60 seconds.

of programs computing the Fibonacci numbers, the square of a number, and the triangular numbers (for non-negative input values). The FUN category consists of properties stating that, under some given preconditions, some of the outputs of the given programs are functionally dependent on a *proper subset* of the inputs.

The results of our experimental evaluation are summarized in Table 1. All experiments have been performed on a single core of an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory running Ubuntu. Timings are in seconds. A time limit of 60 seconds has been set for all problems.

The first two columns (*Cat*) and (*n*) report the names of the categories and the number of problems in each category, respectively. The third column (*Tr*) reports the average time taken for generating the set of CHC encoding the relational property by applying the method presented in Section 3 (including CHC specialization). Columns 4 (*Eld*) and 5 (*Z3*) report the number of problems that were solved by applying Eldarica and Z3, respectively, on the CHC encoding of the relational property. Between parentheses we have indicated the average time taken for each solved problem. There is no column for MathSAT because it is unable to deal with clauses containing multiple atoms in their premise. Column 6 (*PP*) reports the average time taken for applying the predicate pairing transformation strategy presented in Section 4.2. Predicate pairing terminates before the timeout for all problems. Columns 7 (*Eld*), 8 (*MS*), and 9 (*Z3*) report the results obtained by applying Eldarica, MathSAT, and Z3, respectively, on the CHC derived by applying the predicate pairing strategy. We indicate the number of solved problems and the average solving time[5].

If a CHC solver is unable to solve a problem after predicate pairing, we apply CHC specialization again with the goal of propagating the constraints occurring in the clauses and discovering invariants by means of the widening and convex-hull operators [13]. In some cases this constraint propagation produces a set of CHC without constrained facts (that is, clauses of the form $A \leftarrow c$), and hence satisfiable. If the other cases, we apply again the solvers on the clauses obtained after constraint propagation. Columns 10 (*CP+Eld*), 11 (*CP+MS*), and 12 (*CP+Z3*) report the results obtained by applying constraint propagation,

[5] More details about the results of our experimental evaluation are reported in the online appendix available at `http://map.uniroma2.it/cilc16/appendix.pdf`

possibly followed by Eldarica, MathSAT, and Z3, respectively, in the cases where the predicate pairing and the CHC solvers were unable to deliver an answer. As usual, we report the number of problems solved and the average solving time. The solving times include the time spent for constraint propagation.

The use of the predicate pairing and constraint propagation strategies significantly increases the number of problems that have been solved. In particular, the number of problems that can be solved by Eldarica increases from 24 (see Column 4) to 64 (sum of Columns 7 and 10). Similarly for MathSAT from zero to 78 (sum of Columns 8 and 11), and for Z3 from 25 (see Column 5) to 78 (sum of Columns 9 and 12). The number of problems that can be solved by using *any* of the considered CHC solvers is 86.

We observe that the application of the predicate pairing strategy is very effective in increasing the number of solved problems. For instance, it allows Eldarica to solve 15 more problems (see Columns 4 and 7). For problems that can already be solved without using the pairing strategy the performance overhead is almost always tolerable. However, for three problems out of the 90 we have considered, Z3 is unable to prove the property within the considered time limit after the application of the pairing strategy (although one of these problems can be solved after the constraint propagation phase).

Also the application of constraint propagation turns out to be very useful for solving additional problems. For instance, for Z3 constraint propagation allows the solution of additional 44 problems (see Column 12).

It is worth noticing that for the problems in the ITE and REC categories our verification method is competitive with the approach to the proof of program equivalences presented in [21]. As regards the categories MON, FUN, ARR, and LEQ, consist of pairs of programs with a similar control structure and therefore the approach presented in [21] is generally expected to perform well. However, that approach cannot be directly applied to problems in the I-R category).

6 Related Work

Various logics and methods for reasoning about program relations have been presented in the literature. Their main purpose is the formal, possibly automated, validation of program transformation and program analysis techniques.

A Hoare-like axiomatization of relational reasoning for simple while programs has been proposed in [6], which however does not present any technique for automating proofs.

A partial automation of relational reasoning has been proposed in [5], which introduces a notion of a *program product* that allows the reduction of a relational verification problem to a standard program verification problem. The method requires human ingenuity to generate program products via *ad-hoc* refinements and also to provide suitable invariants to the program verifier. Similarly to [5], the *Differential Assertion Checking* technique proposed in [29] makes use of the notion of a *program composition* to reduce the *relative correctness* of two programs to a suitable safety property of the composed program.

The idea of using program transformations to help the proof of relational properties between higher-order functional programs has been explored in [3]. The main difference between the approach in [3] and ours is that, besides the difference of programming languages, we transform the logical representation of the property to be proved, rather than the two programs under analysis. Our approach allows higher parametericity with respect to the programming language, and also enables us to use very effective tools for CHC solving.

Our notion of the predicate pairing is related to that of the *mutual summaries* presented in [24]. Mutual summaries relate the summaries of two procedures, and can be used to prove relations between them, including relative termination (which we do not consider in our technique). Similarly to the above mentioned papers [5,29], this approach requires human ingenuity to generate suitable proof obligations, which can then be discharged by automated theorem provers.

Program equivalence is one of the relational properties that has been extensively studied (see [10,12,21,23,34] for some recent work). Indeed, during software development one may want to modify the program text and prove that its semantics has not changed (this kind of proofs is sometimes called *regression verification*). Among the various approaches to prove program equivalence, the one which is most related to ours is the one reported in [21], which proposes proof rules for the equivalence of simple imperative programs that are then translated into constrained Horn clauses. The satisfiability of these clauses is then checked by state-of-the-art CHC solvers. However, our method can be applied to a larger class of programs without requiring any special purpose proof rules.

Finally, we want to mention that in the present paper we have used (variants and extensions of) transformation techniques for constrained Horn clauses proposed in previous work in the area of program verification (see, for instance, [1,13,14,16,17,28,31]). However, the goal of that previous work was the verification of partial and total correctness of single programs, and not the verification of relations between two programs which has been the objective of our study in this paper.

7 Conclusions

We have presented a method for verifying relational properties of programs written in a simple imperative language with integer and array variables. The method consists in: (i) translating the property to be verified into a set of constrained Horn clauses, then (ii) transforming these clauses to better exploit the interactions between the predicates which represent the computations evoked by the programs, and finally, (iii) using state-of-the-art constrained Horn clause solvers to prove satisfiability.

Although we have considered imperative programs, the only language-specific element of our method is the constrained Horn clause interpreter that we have used to represent in clausal form the program semantics and the property to be verified. Indeed, our method can also be applied to prove relations between programs written in different programming languages. Thus, our approach is basically independent of the programming language used.

Finally, we think that our approach can be refined and improved by taking advantage of the progress that in the future could be made in the development of techniques and tools for reasoning about constrained Horn clauses.

Acknowledgements

We wish to thank Arie Gurfinkel, Vladimir Klebanov, and the participants in the VPT workshop at ETAPS 2015 for stimulating conversations. We acknowledge the financial support of INdAM-GNCS (Italy).

References

1. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. In *Proc. PADL* '07, LNCS 4354, pages 124–139. Springer, 2007.
2. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *Proc. TACAS* '14, LNCS 8413, pages 15–30. Springer, 2014.
3. K. Asada, R. Sato, and N. Kobayashi. Verifying relational properties of functional programs by first-order refinement. In *Proc. PEPM '15*, pages 61–72. ACM, 2015.
4. M. Barnett, B.-Y. E. Chang, R. De Line, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proc. FMCO '06*, LNCS 4111, pages 364–387. Springer, 2006.
5. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Proc. FM '11*, LNCS 6664, pages 200–214. Springer, 2011.
6. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. POPL '04*, pages 14–25. ACM, 2004.
7. N. Bjørner, A. Gurfinkel, K. L. McMillan, and A. Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II - Essays dedicated to Y. Gurevich*, LNCS 9300, pages 24–51. Springer, 2015.
8. C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *J. Autom. Reasoning*, 48(1):107–131, 2012.
9. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proc. VMCAI '06*, LNCS 3855, pages 427–442. Springer, 2006.
10. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *Proc. VMCAI'12*, LNCS 7148, pages 119–135. Springer, 2012.
11. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, eds., *Proc. TACAS '13*, LNCS 7795, pages 93–107. Springer, 2013.
12. S. Ciobâcă, D. Lucanu, V. Rusu, and G. Rosu. A language-independent proof system for mutual program equivalence. In *Proc. ICFEM '14*, LNCS 8829, pages 75–90. Springer, 2014.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95, Part 2:149–175, 2014.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying array programs by transforming verification conditions. In *Proc. VMCAI '14*, LNCS 8318, pages 182–202. Springer, 2014.

15. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *Proc. TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014. `http://www.map.uniroma2.it/VeriMAP`.
16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Proving correctness of imperative programs by linearizing constrained Horn clauses. *Theory and Practice of Logic Programming*, 15(4-5):635–650, 2015.
17. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *Proc. PPDP '15*, pages 91–102. ACM, 2015.
18. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340. Springer, 2008.
19. D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.
20. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
21. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Proc. ASE '14*, pages 349–360, 2014.
22. J.-C. Filliâtre and A. Paskevich. Why3 - Where programs meet provers. In *ESOP '13*, LNCS 7792, pages 125–128. Springer, 2013.
23. B. Godlin and O. Strichman. Regression verification: Proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.
24. C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo. Towards modularly comparing programs using automated theorem provers. In *Proc. CADE 24*, LNCS 7898, pages 282–299. Springer, 2013.
25. H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, and P. Rümmer. A verification toolkit for numerical transition systems. In *Proc. FM '12*, LNCS 7436, pages 247–251. Springer, 2012.
26. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
27. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
28. B. Kafle and J. P. Gallagher. Constraint specialisation in Horn clause verification. In *Proc. PEPM '15*, pages 85–90. ACM, 2015.
29. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proc. ESEC/FSE '13*, pages 345–355. ACM, 2013.
30. Y. V. Matijasevic. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR* (*in Russian*), 191:279–282, 1970.
31. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. In *Proc. SAS '98*, LNCS 1503, pages 246–261. Springer, 1998.
32. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
33. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. ICLP '84*, pages 127–138, 1984.
34. S. Verdoolaege, G. Janssens, and M. Bruynooghe. Equivalence checking of static affine programs using widening to handle recurrences. *ACM Trans. Program. Lang. Syst.*, 34(3):11, 2012.
35. A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proc. FM '08*, LNCS 5014, pages 35–51. Springer, 2008.