Relational Verification through Horn Clause Transformation

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 operational semantics of the programming language in which the two programs are written. That definition of the semantics consists of a set *Int* of constrained Horn clauses (CHCs) 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 CHCs 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 unfold/fold rules, preserve satisfiability. Through an experimental evaluation, we show that in many cases CHC solvers are able to prove the satisfiability (or the unsatisfiability) 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, untransformed sets of CHCs.

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 program fragment 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. Among the many examples of relational properties that can be considered in practice as indicated in some papers [5,30], *program equivalence* has a prominent significance.

It has been noted that proving relational properties between two structurally similar program versions is often easier than directly proving the desired correctness property for the new program version [5,20,30]. Moreover, in order to automate the correctness proofs, it may be convenient to follow a transformational approach so that one can use the already available methods and tools for proving correctness of individual programs. For instance, various papers 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 [5,30,38]. The validity of these pre- and post-conditions is then checked by using state-of-the-art verification tools (e.g., Boogie [4] and Why [21]). A different transformational approach is followed by Felsing et al. [20], who 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 verification techniques that 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 [26] and Z3 [17] (obviously, no complete solver exists because most properties of interest, and among them equivalence, are in general undecidable). Unfortunately, all the above mentioned approaches enable only a partial reuse of the available verification techniques, because one should develop specific transformation rules for each programming language and each proof system that has to be used.

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 [16,34]. 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 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 mentioned, 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 component of our method is 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 in use. 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 [19,36].
(3) We report on an experimental evaluation performed by a proof-of-concept implementation (see Figure 1) of our transformation technique by using the VeriMAP system [14]. The satisfiability of the transformed CHCs is then verified by using the solvers Eldarica [26] and Z3 [17]. Our experiments show that the transformation is effective on a number of small, yet nontrivial examples. Moreover, our method is competitive with respect to other tools for checking relational properties [20].

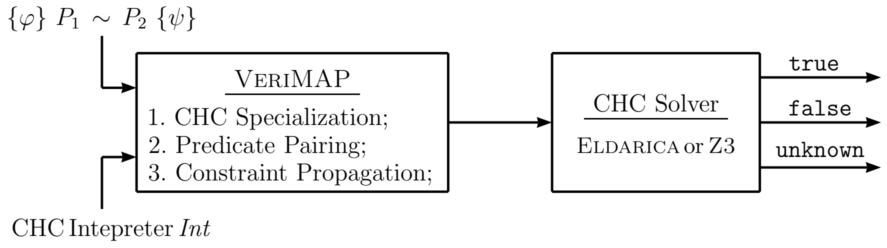

Fig. 1. The verification method. The VeriMAP system transforms CHCs, and the CHC solver checks the satisfiability of CHCs. VeriMAP takes as input: (i) the CHC encoding of a relational property $\{\varphi\}\, P_1 \sim P_2\, \{\psi\}$ between programs P_1 and P_2, and (ii) the CHC interpreter that encodes the semantics of the programming language.

The paper is organized as follows. In Section 2 we present a simple introductory example. Then, in Section 3 we present the method for translating a relational property between two programs into constrained Horn clauses. In Section 4 we introduce the transformation techniques for CHCs and we prove that they preserve satisfiability (and unsatisfiability). The implementation of the verification method and its experimental evaluation are 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 approach proposed in this paper. Let us consider the two imperative programs of Figure 2. Program `sum_upto` computes the sum of the first `x1` non-negative integers and program prod computes the product of x2 by y2 by summing up x2 times the value of y2. We have that the following relational property holds:

Leq: {x1 = x2, x2 ≤ y2} sum_upto ∼ prod {z1 ≤ z2}

meaning that, if x1 = x2 and x2 ≤ y2 hold before the execution of sum_upto and prod, then z1 ≤ z2 holds after their execution. Property *Leq* cannot directly be proved using techniques based on structural similarities of programs [5,20], because sum_upto is a non-tail recursive program and prod is an iterative program.

```
/* -- Program sum_upto -- */

int x1, z1;

int f(int n1){
  int r1;
  if (n1 <= 0) {
    r1 = 0;
  } else {
    r1 = f(n1 - 1) + n1; }
  return r1; }

void sum_upto() {
  z1 = f(x1); }
```

```
/* -- Program prod -- */

int x2,y2,z2;

int g(int n2, int m2){
  int r2;
  r2 = 0;
  while (n2 > 0) {
    r2 += m2;
    n2--; }
  return r2; }

void prod() {
  z2 = g(x2,y2); }
```

Fig. 2. The programs sum_upto and prod.

By using the method presented in Section 3 and the CHC Specialization presented in Section 4.1, the relational property *Leq* is translated into the set of constrained Horn clauses shown in Figure 3.

A *constrained Horn clause* is an implication of the form: $A_0 \leftarrow c, A_1, \ldots, A_n$, where: (i) A_0 is either an *atomic formula* (or an *atom*, for short) or *false*, (ii) c is a *constraint*, that is, a quantifier-free formula of the theory of linear integer arithmetic and integer arrays [9], and (iii) $A_1, \ldots, A_n$ is a possibly empty conjunction of atoms. A_0 is said to be the *head* of the clause, and the conjunction $c, A_1, \ldots, A_n$ is said to be the *body* of the clause. If the body is the empty conjunction of constraints and atoms (i.e., *true*), then the clause is called a *fact*.

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

Fig. 3. Translation into constrained Horn clauses of the relational property *Leq*.

Clause 1 specifies the relational property *Leq*, where the logical variables $Z1$ and $Z2$ refer to the values of the imperative variables z1 and z2, respectively. Note that the constraint $Z1 > Z2$ in the body of clause 1 encodes the negation of the postcondition z1 ≤ z2 we want to prove. Clauses 2–4 and 5–7 encode the input-output relations computed by programs sum_upto and prod, respectively. We have that 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 [26] and Z3 [17]) are unable to prove the satisfiability of clauses 1–7. This is due to the fact that those solvers act on the predicates *su* and *pr* 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, in general, undecidable [31].

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, for which the satisfiability (or unsatisfiability) is hopefully easier to prove. In our example, the clauses of Figure 3 are transformed into the ones shown in Figure 4.

10. $\mathit{false} \leftarrow N \leq Y,\ W=0,\ Z1 > Z2,\ \mathit{fg}(N, Z1, Y, W, Z2)$
11. $\mathit{fg}(N, Z1, Y, Z2, Z2) \leftarrow N \leq 0,\ Z1=0$
13. $\mathit{fg}(N, Z1, Y, W, Z2) \leftarrow N \geq 1,\ N1=N-1,\ Z1=R+N,\ M=Y+W,$
    $\mathit{fg}(N1, R, Y, M, Z2)$

Fig. 4. Transformed clauses derived from the clauses 1–7 in Figure 3. Clause numbers are those indicated in the derivation of Section 4.2.

The predicate $\mathit{fg}(N, Z1, Y, W, Z2)$ is equivalent to the conjunction '$f(N, Z1)$, $g(N, Y, W, Z2)$'. The effect of this transformation is that it is possible to infer linear relations among a subset of the variables occurring in *conjunctions* of predicates, without having to use in an explicit way their non-linear relations with other variables. In particular, one can infer that, whenever $W=0$, $\mathit{fg}(N, Z1, Y, W, Z2)$ enforces the constraint $(N > Y) \vee (Z1 \leq Z2)$, and hence the satisfiability of clause 10 of Figure 4, without having to derive quadratic relations. Indeed, after this transformation, state-of-the-art solvers for CHCs with linear arithmetics are able to prove the satisfiability of the clauses of Figure 4, which implies the validity of the relational property *Leq* (see Section 5).

3 Specifying Relational Properties using CHCs

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 CHCs.

3.1 Relational properties

We consider a C-like imperative programming language manipulating integers and integer arrays via assignments, function calls, conditionals, while-loops, and jumps. A program is a sequence of labeled commands (or commands, for short). We assume that in each program there is a unique `halt` command that, when executed, causes program termination. We will feel free to write commands without their labels and programs without their `halt` commands, whenever those labels and commands are not needed for specifying the semantics of the programs.

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 a variable identifier x to its value v *either* in the integers (for integer variables) *or* in the set of 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 a command, different from 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 $\Longrightarrow^*$ of $\Longrightarrow$.

In particular, the semantics of an assignment is:
$(R1)\quad \langle\!\langle \ell\!:\!x\!=\!e,\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(nextlab(\ell)),\ update(\delta, x, [\![e]\!]\,\delta) \rangle\!\rangle$
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 δ.

The semantics of a call to the function f is:
$(R2)\quad \langle\!\langle \ell\!:\!x\!=\!f(e_1,\ldots,e_k),\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(nextlab(\ell)), update(\delta', x, [\![e]\!]\,\delta') \rangle\!\rangle$
$\qquad\qquad$ if $\langle\!\langle at(firstlab(f)),\ \overline{\delta} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_r\!: \texttt{return}\ e,\ \delta' \rangle\!\rangle$
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 [16] for a more detailed presentation of the multistep semantics.

A program P *terminates* for an initial environment δ, whose domain includes all global variables of P, and computes the final environment η, denoted $\langle P, \delta \rangle \Downarrow \eta$, iff $\langle\!\langle \ell_0 : c, \delta \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \texttt{halt}, \eta \rangle\!\rangle$, 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. It follows from the definition of the operational semantics that also the domain of η includes all global variables of P.

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 Barthe et al. [5], a relational property is specified by the 4-tuple $\{\varphi\}\ P_1 \sim P_2\ \{\psi\}$. (See property *Leq* in Section 2 for an example.)

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 χ and an environment δ, by $\chi\,[\delta]$ we denote the formula χ where every free occurrence of a variable x in $dom(\delta)$ has been replaced by $\delta(x)$.

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* $dom(\delta_1) = \mathcal{V}_1$ *and* $dom(\delta_2) = \mathcal{V}_2$*, the following holds*:

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

3.2 Formal Semantics of the Imperative Language in CHCs

In order to translate a relational program property into CHCs, first we need to specify the operational semantics of our imperative language by a set of CHCs. We follow the approach proposed by De Angelis et al. [16] which now we briefly recall. The transition relation $\Longrightarrow$ between configurations and its reflexive, transitive closure $\Longrightarrow^*$ 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 the original paper [16].

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

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

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

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

A program is represented by a set of facts of the form $at(L, C)$, where L and C encode a label and a command, respectively. For instance, program `sum_upto` of Figure 2 is represented by the following facts:

$at(0, ite(lte(\mathtt{n1}, \mathtt{0}), 1, 2)) \leftarrow$
$at(1, asgn(\mathtt{r1}, \mathtt{0})) \leftarrow$
$at(2, asgn(\mathtt{r1}, call(\mathtt{f}, [minus(\mathtt{n1}, \mathtt{1})]))) \leftarrow$
$at(3, asgn(\mathtt{r1}, plus(\mathtt{r1}, \mathtt{n1}))) \leftarrow$
$at(4, return(\mathtt{r1})) \leftarrow$
$at(5, asgn(\mathtt{z1}, call(\mathtt{f}, [\mathtt{x1}]))) \leftarrow$
$at(6, \mathtt{halt}) \leftarrow$

In this representation of the program `sum_upto` the `halt` command and the labels of the commands, which were omitted in the listing of Figure 2, have been explicitly shown. Configurations are represented by terms of the form $cf(cmd(L, C), Env)$, where: (i) $cmd(L, C)$ encodes a command C with label L, and (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 $R2$ (see Section 3.1).

Given a program *Prog*, its input-output relation is represented by a predicate *prog* defined as follows:

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

where $\mathit{initConf}(C,X)$ and $\mathit{finalConf}(C',X')$ 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 CHCs

In any given a relational property $\{\varphi\}\, P_1 \sim P_2\, \{\psi\}$ between programs $P1$ and $P2$, we assume that φ and ψ are constraints, that is, as mentioned in Section 2, quantifier-free formulas of the theory of linear integer arithmetic and integer arrays [9]. Let $\mathcal{A}$ be the set of axioms of this theory. The set $\mathcal{C}$ of constraints is closed under conjunction and, when writing clauses, we will use comma to denote a conjunction of constraints.

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).

A set S of CHC clauses is said to be $\mathcal{A}$-*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 CHC clause:

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

where: (i) X and Y are the disjoint tuples of the global variables $\mathtt{x_i}$'s of program P_1 and the global variables $\mathtt{y_i}$'s of program P_2, respectively, rewritten in capital letters (so to comply with CHC syntax);
(ii) $\mathit{pre}(X,Y)$ is the formula φ with its variables replaced by the corresponding capital letters;
(iii) $\mathit{neg\_post}(X',Y')$ is the formula $\neg\psi$ with its variables replaced by the corresponding primed capital letters; and
(iv) the predicates $p1(X,X')$ and $p2(Y,Y')$ are defined by a set of clauses derived from program P_1 and P_2, respectively, by using the formalization of the operational semantics of the programming language presented in Section 3.2.

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 two or more clauses with *false* in their head. Although these transformations are not strictly needed by the techniques described in the rest of the paper, they are useful when automating our verification method using constraint solving tools.

For instance, the relational property *Leq* of Section 2 is translated into the following clause (we moved all constraints to leftmost positions in the body):

$$(\mathit{Prop}_{\mathit{Leq}}) \quad \mathit{false} \leftarrow X1\!=\!X2,\ X2\!\leq\!Y2,\ Z1'\!>\!Z2',\\ \mathit{sum\_upto}(X1,Z1,X1',Z1'),\ \mathit{prod}(X2,Y2,Z2,X2',Y2',Z2')$$

where predicates *sum_upto* and *prod* are defined in terms of the predicate *reach* shown in Section 3.2. In particular, *sum_upto* is defined by the following clauses:

$$\mathit{sum\_upto}(X1,Z1,X1',Z1') \leftarrow\\ \mathit{initConf}(C,X1,Z1), \mathit{reach}(C,C'), \mathit{finalConf}(C',X1',Z1')$$
$$\mathit{initConf}(\mathit{cf}(\mathit{cmd}(5,\mathit{asgn}(\mathtt{z1},\mathit{call}(\mathtt{f},[\mathtt{x1}]))),[(\mathtt{x1},X1),(\mathtt{z1},Z1)]),X1,Z1) \leftarrow$$
$$\mathit{finalConf}(\mathit{cf}(\mathit{cmd}(6,\mathtt{halt}),[(\mathtt{x1},X1'),(\mathtt{z1},Z1')]),X1',Z1') \leftarrow$$

where: (i) *initConf* holds for the initial configuration of the `sum_upto` program (that is, the pair of the assignment `z1 = f(x1)` and the initial environment), and (ii) *finalConf* holds for the final configuration of the `sum_upto` program (that is, the pair of the unwritten `halt` command silently occurring after the assignment `z1 = f(x1)` and the final environment).

Let *RP* be a relational property and T_{RP} be the set of CHCs generated from *RP* by the translation process described above, then T_{RP} is correct in the sense specified by the following theorem.

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 *tr* is a correct formalization of the semantics of the programming language.

4 Transforming Specifications of Relational Properties

The reduction of the problem of checking 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. Indeed, we can try to solve the satisfiability problem for T_{RP} by applying the available solvers for constrained Horn clauses. Unfortunately, as shown by the example in Section 2, it may be the case that these solvers are unable 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 to show the satisfiability of the set T_{RP} of constrained Horn clauses is that we can transform T_{RP} by applying any CHC satisfiability preserving algorithm, and then submit the transformed clauses to a CHC solver.

In this section we present two satisfiability preserving algorithms for transforming constrained Horn clauses that have the objective of increasing the effectiveness of the subsequent uses of CHC solvers. These algorithms, called *transformation strategies*, are: (1) the *CHC Specialization*, and (2) the *Predicate Pairing*.

These strategies are variants of techniques developed in the area of logic programming for improving the efficiency of program execution [18,35]. In Section 5 we will give an experimental evidence that these techniques are very effective for the verification of relational properties. The 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 a couple of decades ago [19]. Now we present the version of those rules we need in our context. This version allows us to derive from an old set *Cls* of constrained Horn clauses a new set *TransfCls* of constrained Horn clauses.

The *definition* rule allows us to introduce a new predicate symbol in *Cls* defined by a single clause.

Definition Rule. We introduce a *definition clause* D of the form $\mathit{newp}(X) \leftarrow c, G$, 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 definition clauses introduced in a sequence of application of the unfold/fold rules.

The *unfolding* rule consists in applying a resolution step with respect to an atom selected in the body of a clause in *Cls*.

Unfolding Rule. Let C be a clause 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 $i = 1, \ldots, m$, the following two conditions hold: (1) A is unifiable with K_i via the most general unifier ϑ_i, and (2) $\mathcal{A} \models \exists X\, (c\,, c_i)\, \vartheta_i$, where X is the tuple of variables in $(c\,, c_i)\, \vartheta_i$ (recall that by comma we denote conjunction). 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 $\{(H \leftarrow c, c_i, L, B_i, R)\, \vartheta_i \mid i = 1, \ldots, m\}$.

Since the definition rule introduces new predicates through a single definition clause, the head and the body of that clause are equivalent in the least $\mathcal{C}$-model of *Cls* [27]. The *folding* rule consists in replacing an instance of the body of a definition clause by the corresponding instance of its head.

Folding Rule. Let E be a clause in *Cls* of the form: $H \leftarrow e, L, Q, R$. Suppose that there exists a clause D in *Defs* of the form $K \leftarrow d, G$ such that: (1) for some substitution ϑ, $Q = G\,\vartheta$, and (2) $\mathcal{A} \models \forall X\ (e \rightarrow d\,\vartheta)$ 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 of Etalle and Gabbrielli [19], which ensure that the transformation rules preserve the least $\mathcal{C}$-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.*

Note that the applicability condition of Theorem 2 avoids that an unsatisfiable set of clauses is transformed into a satisfiable set. For instance, the unsatisfiable set of clauses $\{\, \mathit{false} \leftarrow p,\;\; p \leftarrow \,\}$ could be transformed into the satisfiable set of clauses (by taking q to be *false* and p to be *true*) $\{\, \mathit{false} \leftarrow q,\;\; q \leftarrow q,\;\; p \leftarrow \,\}$, by first introducing the new definition clause $q \leftarrow p$ and then folding both $\mathit{false} \leftarrow p$ and $q \leftarrow p$, using $q \leftarrow p$. This sequence of applications of the rules is avoided by the condition that the definition $q \leftarrow p$ must be unfolded, and hence replaced by the fact q.

4.1 CHC Specialization

Specialization is a transformation technique that has been proposed in various programming contexts to take advantage of static information for simplifying and customizing programs [28]. In the field of program verification it has been shown that the specialization of constrained Horn clauses can be very useful for transforming and simplifying clauses before checking their satisfiability [12,16,29].

We will use specialization of constrained Horn clauses, called *CHC Specialization*, for simplifying the set T_{RP} of clauses. In particular, starting from clause *Prop* (see Section 3.3):

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

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 $\mathit{pre}(X,Y)$ or $\mathit{neg\_post}(X',Y')$. Then, by applying the folding rule to clause *Prop*, we can replace $p1$ and $p2$ by $p1_{sp}$ and $p2_{sp}$, respectively, thereby obtaining:

$(\mathit{Prop}_{sp})\qquad \mathit{false} \leftarrow \mathit{pre}(V,W),\; p1_{sp}(V,V'),\; p2_{sp}(W,W'),\; \mathit{neg\_post}(V',W')$

Now, by applying the specialization strategy proposed by De Angelis et al. [16], starting from the set $(T_{RP}-\{\mathit{Prop}\}) \cup \{\mathit{Prop}_{sp}, S1, S2\}$ of clauses, we will derive specialized versions of the clauses that define the semantics of programs P_1 and P_2. In particular, in the specialized clauses there will be references to neither the predicate *reach*, nor the predicate *tr*, nor the terms encoding configurations.

Let us illustrate the application of the specialization strategy [16] by considering again the example of Section 2. Starting from clause Prop_{Leq} of Section 3.3, that is:

$(\mathit{Prop}_{Leq})\quad \mathit{false} \leftarrow X1 = X2,\;\; X2 \leq Y2,\;\; Z1' > Z2',$
$\qquad\qquad\qquad \mathit{sum\_upto}(X1, Z1, X1', Z1'),\;\; \mathit{prod}(X2, Y2, Z2, X2', Y2', Z2')$

we introduce two new predicates, namely *su* and *pr*, defined as follows:

$(S1_{Leq})\;\; \mathit{su}(X1, Z1') \leftarrow \mathit{sum\_upto}(X1, Z1, X1', Z1')$

$(S2_{Leq})\;\; \mathit{pr}(X2, Y2, Z2') \leftarrow \mathit{prod}(X2, Y2, Z2, X2', Y2', Z2')$

By applying the folding rule to Prop_{Leq}, we replace $\mathit{sum\_upto}(X1, Z1, X1', Z1')$ and $\mathit{prod}(X2, Y2, Z2, X2', Y2', Z2')$ by $\mathit{su}(X1, Z1')$ and $\mathit{pr}(X2, Y2, Z2')$, respectively, and we get (modulo variable renaming) clause 1 of Figure 3.

Then, starting from $S1_{Leq}$ and $S2_{Leq}\}$ we get clauses 2–7 of Figure 3 as we now show. We only illustrate the first few steps of the application of the specialization strategy, which consists in applying a sequence of the transformation rules presented in Section 4.

Unfolding: The strategy starts by unfolding $\mathit{sum\_upto}(X1, Z1, X1', Z1')$ occurring in the body of the clause $S1_{Leq}$, hence deriving:

$$\mathit{su}(X1, Z1') \leftarrow \mathit{initConf}(C, X1, Z1), \mathit{reach}(C, C'), \mathit{finalConf}(C', X1', Z1')$$

By unfolding the above clause w.r.t. *initConf*, *reach*, and *finalConf*, we get:

$$\begin{aligned}\mathit{su}(X1, Z1') \leftarrow\ & \mathit{tr}(\mathit{cf}(\mathit{cmd}(5, \mathit{asgn}(\mathtt{z1}, \mathit{call}(\mathtt{f}, [\mathtt{x1}]))), [(\mathtt{x1}, X1),(\mathtt{z1}, Z1)]), C),\\ & \mathit{reach}(C,\mathit{cf}(\mathit{cmd}(6, \mathtt{halt},[(\mathtt{x1}, X1'),(\mathtt{z1}, Z1')]))\end{aligned}$$

Then, by unfolding the above clause w.r.t. *tr*, we get:

$$\begin{aligned}&\mathit{su}(X1, Z1') \leftarrow\\ &\quad \mathit{reach}(\mathit{cf}(\mathit{cmd}(0, \mathit{ite}(\mathit{lte}(\mathtt{n1},\mathtt{0}), 1, 2), [(\mathtt{x1}, X1),(\mathtt{z1}, Z1), (\mathtt{n1}, N1),(\mathtt{r1}, R1)]),\\ &\qquad \mathit{cf}(\mathit{cmd}(6, \mathtt{halt},[(\mathtt{x1}, X1'),(\mathtt{z1}, Z1')]))\end{aligned}$$

Definition and Folding: By the definition rule we introduce the following clause defining the new predicate f:

$$\begin{aligned}&f(X1, Z1') \leftarrow\\ &\quad \mathit{reach}(\mathit{cf}(\mathit{cmd}(0, \mathit{ite}(\mathit{lte}(\mathtt{n1},\mathtt{0}), 1, 2), [(\mathtt{x1}, X1),(\mathtt{z1}, Z1), (\mathtt{n1}, N1),(\mathtt{r1}, R1)]),\\ &\qquad \mathit{cf}(\mathit{cmd}(6, \mathtt{halt},[(\mathtt{x1}, X1'),(\mathtt{z1}, Z1')]))\end{aligned}$$

Then, the new definition can used for folding, hence deriving (modulo variable renaming) clause 2 of Figure 3.

Starting from the new definition for predicate *f*, after a similar sequence of applications of the unfolding, definition, and folding rules, we get clauses 3–4. Then, if we perform for the predicate *pr* the analogous transformation steps we have done for the predicate *sum_upto*, that is, if we start from clause $S2_{Leq}$, instead of $S1_{Leq}$, we eventually get the other clauses 5–7 of Figure 3.

Since the CHC Specialization is performed by applying the unfold/fold rules, by Theorem 2 we have the following result.

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 second strategy which characterizes our verification method is the Predicate Pairing strategy (see Figure 5). This strategy pairs together two predicates, say q and r into one new predicate t equivalent to their conjunction. As shown in the example of Section 2, Predicate Pairing may ease the discovery of relations among the arguments of the two distinct predicates, and thus it may ease the satisfiability test. Obviously, pairing may be iterated and more than two predicates may in general be tupled together.

Let us see the Predicate Pairing strategy in action by considering again the example of Section 2.

First Iteration of the while-loop of the Predicate Pairing strategy.
UNFOLDING: The strategy starts by unfolding $\mathit{su}(X1, Z1)$ and $\mathit{pr}(X2, Y2, Z2)$ in clause 1 of Figure 3, hence deriving the following new clause:

8. $\mathit{false} \leftarrow N \leq Y,\ W{=}0,\ Z1 > Z2,\ f(N, Z1),\ g(N, Y, W, Z2)$

```
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 $\mathit{TransfCls}$ of clauses.

INITIALIZATION:   $\mathit{InCls} := \{C\}$;   $\mathit{Defs} := \emptyset$;   $\mathit{TransfCls} := Q \cup R$;
while there is a clause $C$ in $\mathit{InCls}$ 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 $Q \cup R$;
    Perform as long as possible the following replacement: for every clause $C'$ of the
    form: $H \leftarrow d,\ A_1, ..., A_k$ in $U(C)$, for every pair of (not necessarily distinct) atoms
    $A_i = p_i(...,X,...)$ and $A_j = p_j(...,Y,...)$, if $\mathcal{A} \models \forall(d \rightarrow (X=Y))$, then replace every
    occurrence of $Y$ by $X$ in $C'$, thereby updating the set $U(C)$ of clauses.
  DEFINITION & FOLDING:
    $F(C) := U(C)$;
    for every clause $E \in F(C)$ of the form $H \leftarrow d,\ G_1, q(V), G_2, r(W), G_3$ where $q$
        and $r$ occur in $Q$ and $R$, respectively, do
      if in $\mathit{Defs}$ there is no clause $D$ of the form $\mathit{newp}(Z) \leftarrow q(V), r(W)$ (modulo
          variable renaming), where $Z$ is the tuple of distinct variables in $(V, W)$
      then  $\mathit{InCls} := \mathit{InCls} \cup \{D\}$;   $\mathit{Defs} := \mathit{Defs} \cup \{D\}$;
      $F(C) := (F(C) - \{E\}) \ \cup\ \{H \leftarrow d,\ G_1, \mathit{newp}(Z), G_2, G_3\}$
    end-for
  $\mathit{InCls} := \mathit{InCls} - \{C\}$;   $\mathit{TransfCls} := \mathit{TransfCls} \cup F(C)$;
end-while
```

Fig. 5. The *Predicate Pairing* transformation strategy.

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

9. $\mathit{fg}(N, Z1, Y, W, Z2) \leftarrow f(N, Z1),\ g(N, Y, W, Z2)$

and that conjunction is folded, hence deriving:

10. $\mathit{false} \leftarrow N \leq Y,\ W=0,\ Z1 > Z2,\ \mathit{fg}(N, Z1, Y, W, Z2)$

Second Iteration of the while-loop of the Predicate Pairing strategy.
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. $\mathit{fg}(N, Z1, Y, Z2, Z2) \leftarrow N \leq 0,\ Z1=0$
12. $\mathit{fg}(N, Z1, Y, W, Z2) \leftarrow N \geq 1,\ N1=N-1,\ Z1=R+N,\ M=Y+W,$
    $f(N1, R),\ g(N1, Y, M, Z2)$

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. $\mathit{fg}(N, Z1, Y, W, Z2) \leftarrow N \geq 1,\ N1=N-1,\ Z1=R+N,\ M=Y+W,$
    $\mathit{fg}(N1, R, Y, M, Z2)$

Clauses 10, 11, and 13, which are the ones shown in Figure 4, 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 k of conjunctions of the form $q(V), r(W)$, where q occurs in Q and r occurs in R and, hence, the number of executions of the while-loop of the strategy is bounded by k.

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

Theorem 4 (Termination and 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.*

4.3 Verifying Loop Composition

Once the relational property has been translated into a set of CHCs, its verification is no longer dependent on the syntax of the source programs. Thus, as already mentioned in Section 2, we may be able to verify relations between programs that have different structure (e.g., relations between non-tail recursive programs and iterative programs). In this section we show one more example of a property that relates two programs that are not structurally similar (in particular, they are obtained by composing different numbers of while-loops).

Let us consider the programs `sum1` and `sum2` in Figure 6. They both compute the sum of all integers up to `m1` and `m2`, respectively. Program `sum1` computes the result using a single while-loop, and `sum2` consists of the composition of two while-loops: the first loop sums the numbers up to an integer `n2`, with $1 \leq \mathtt{n2} < \mathtt{m2}$, and the second loop adds to the result of the first loop the sum of all numbers from `n2+1` up to `m2`. We want to show the following relational property:

LeqS: $\{\mathtt{m1}=\mathtt{m2}, 1 \leq \mathtt{n2} < \mathtt{m2}\}$ `sum1` $\sim$ `sum2` $\{\mathtt{s1} \leq \mathtt{s2}\}$

```
/* -- Program sum1 -- */

int m1, s1;

void sum1() {
  int i1 = 0;
  s1 = 0;

  while (i1 <= m1) {
    s1 += i1;
    i1++; }
```

```
/* -- Program sum2 -- */

int n2, m2, s2;

void sum2() {
  int i2 = 0;
  s2 = 0;

  while (i2 <= n2) {
    s2 += i2;
    i2++; }

  while (i2 <= m2) {
    s2 += i2;
    i2++; } }
```

Fig. 6. The programs `sum1` and `sum2`.

The relational property *LeqS* and the two programs are translated into the set of clauses shown in Figure 7. Neither the solvers for CHCs with linear arithmetics (in particular, we tried Eldarica and Z3) nor Rêve, the tool for relational verification that implements the approach proposed by Felsing et al. [20], are able to prove the property *LeqS*.

The Predicate Pairing strategy transforms the clauses of Figure 7 into the clauses of Figure 8 (the variable names are automatically generated by our implementation). The strategy introduces the two new predicates $s1s2$ and $s1s3$, which stand for the conjunctions of $s1$ with $s2$, and $s1$ with $s3$, respectively.

$\mathit{false} \leftarrow M1\!=\!M2, 1\!\leq\! N2, N2\!<\!M2, S1\!>\!S2, \mathit{sum}1(M1,S1), \mathit{sum}2(M2,N2,S2)$
$\mathit{sum}1(M1,S1) \leftarrow S1i\!=\!0,\ I1\!=\!0,\ s1(M1,S1i,I1,S1)$
$s1(M1,S1i,I1,S1) \leftarrow M1\!<\!I1,\ S1i\!=\!S1$
$s1(M1,S1i,I1,S1) \leftarrow M1\!\geq\! I1,\ T\!=\!S1i\!+\!I1,\ J\!=\!I1\!+\!1,\ s1(M1,T,J,S1)$
$\mathit{sum}2(M2,N2,S2) \leftarrow S2i\!=\!0,\ I2\!=\!0,\ s2(M2,N2,S2i,I2,S2)$
$s2(M2,N2,S2i,I2,S2) \leftarrow M2\!<\!I2,\ s3(N2,S2i,I2,S2)$
$s2(M2,N2,S2i,I2,S2) \leftarrow M2\!\geq\! I2,\ U\!=\!S2i\!+\!I2,\ K\!=\!I2\!+\!1,\ s2(M2,N2,U,K,S2)$
$s3(N2,S2i,I2,S2) \leftarrow N2\!<\!I2,\ S2i\!=\!S2$
$s3(N2,S2i,I2,S2) \leftarrow N2\!\geq\! I2,\ U\!=\!S2i\!+\!I2,\ K\!=\!I2\!+\!1,\ s3(N2,U,K,S2)$

Fig. 7. Translation into constrained Horn clauses of the relational property *LeqS*.

1. $\mathit{false} \leftarrow 1\!\leq\! E,\ E\!<\!A,\ D\!>\!F,\ B\!=\!0,\ C\!=\!0,\ s1s2(A,B,C,D,E,F)$
2. $s1s2(A,B,C,D,E,F) \leftarrow A\!<\!C,\ B\!=\!D,\ E\!<\!C,\ s3(A,B,C,F)$
3. $s1s2(A,B,C,D,E,F) \leftarrow A\!<\!C,\ B\!=\!D,\ E\!\geq\! C,\ G\!=\!B\!+\!C,\ H\!=\!C\!+\!1,$
   $s2(E,A,G,H,F)$
4. $s1s2(A,B,C,D,E,F) \leftarrow A\!\geq\! C,\ E\!<\!C,\ G\!=\!B\!+\!C,\ H\!=\!C\!+\!1,$
   $s1s3(A,G,H,D,B,C,F)$
5. $s1s2(A,B,C,D,E,F) \leftarrow A\!\geq\! C,\ E\!\geq\! C,\ G\!=\!B\!+\!C,\ H\!=\!C\!+\!1,$
   $s1s2(A,G,H,D,E,F)$
6. $s1s3(A,B,C,D,E,F,G) \leftarrow A\!<\!C,\ B\!=\!D,\ A\!<\!F,\ E\!=\!G$
7. $s1s3(A,B,C,D,E,F,G) \leftarrow A\!<\!C, B\!=\!D, A\!\geq\! F, H\!=\!E\!+\!F, I\!=\!E\!+\!1,\ s3(A,H,I,G)$
8. $s1s3(A,B,C,D,E,F,G) \leftarrow A\!\geq\! C, A\!<\!F, E\!=\!G, H\!=\!B\!+\!C, I\!=\!C\!+\!1,\ s1(A,H,I,D)$
9. $s1s3(A,B,C,D,E,F,G) \leftarrow A\!\geq\! C,\ A\!\geq\! F,\ H\!=\!B\!+\!C,\ I\!=\!C\!+\!1,\ J\!=\!E\!+\!F,\ K\!=\!F\!+\!1,$
   $s1s3(A,H,I,D,J,K,G)$

Fig. 8. Output of the Predicate Pairing transformation for property *LeqS*.

The clauses of Figure 8 can be further simplified. For instance, by propagating the constraints that occur in clause 1, we can discover that clause 2 can be removed, because it refers to values of the loop index for which `sum1` has terminated and the second while-loop of `sum2` is still in execution, and this is impossible with the given pre-relation. Similarly, clauses 3, 7, and 8 can be removed. This form of post-processing of the clauses obtained by Predicate Pairing can be performed by *Constraint Propagation*, through the use of CHC specialization [12]. In Section 5 we will demonstrate the positive effects of Constraint Propagation after Predicate Pairing, and indeed the satisfiability of the clauses of Figure 8 is easily proved by Eldarica and Z3 after Constraint Propagation.

5 Implementation and Experimental Evaluation

We have implemented the techniques presented in Sections 3 and 4 as a part of the VeriMAP verification system [14], and we have used the SMT solvers Eldarica and Z3 (collectively called CHC solvers) for checking the satisfiability of the CHCs generated by VeriMAP.

In particular, our implementation consists of the following three modules. (1) A front-end module, based on the C Intermediate Language (CIL) [33], that translates the given verification problem into the facts defining the predicates *at*, *initConf*, and *finalConf*, by using a custom implementation of the CIL visitor pattern. (2) A back-end module realizing the *CHC Specialization* and *Predicate Pairing* transformation strategies. (3) A module that translates the generated CHCs to the SMT-LIB format for the ELDARICA and Z3 solvers.

We have considered 100 problems[4] referring to relational properties of small, yet non-trivial, C programs mostly taken from the literature [5,6,20]. All programs act on integers, except the programs in the ARR category and 7 out of 10 in the CON category which act on integer arrays. The properties we have considered belong to the following categories. 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 output when given the same input. 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 preconditions, 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 of programs computing the Fibonacci numbers, the square of a number, and the triangular numbers. 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 COMP category consists of equivalence and inequality properties relating two programs that contain compositions of different numbers of loops (see the example in Section 4.3).

The experimental process interleaves the application of a CHC transformation strategy (performed by VERIMAP) and a CHC satisfiability check (performed by ELDARICA and Z3). We have considered the following strategies: (i) the CHC Specialization, Sp for short, which is the strategy presented in Section 4.1 that transforms the set of CHCs encoding the relational property, (ii) the Predicate Pairing, PP for short, which is the strategy presented in Section 4.2, and (iii) the Constraint Propagation, CP for short, which is the strategy that propagates the constraints occurring in the clauses with the aim of discovering invariant constraints by means of the widening and convex-hull operators [12]. We have used the following CHC solvers for checking satisfiability of CHCs: (i) ELDARICA (v1.2-rc in standard mode[5]), and (ii) Z3 (version 4.4.2,

[4] The sources of the problems are available at `http://map.uniroma2.it/relprop`

[5] using the options `-horn -hsmt -princess -i -abstract:oct`. Running ELDARICA in client-server mode could significantly improve its performance, but requires custom modifications for running multiple problems in parallel.

Problem Category	M	Sp	PP	CP	(1) Sp + Eld		(2) Sp + Z3		(3) Sp + PP + Eld		(4) Sp + PP + Z3		(5) Sp + PP+CP+Eld		(6) Sp + PP+CP+Z3	
		time	time	time	N	time	N	time	N	time	N	time	N	time	N	time
ITE	21	0.10	5.32	0.46	9	23.94	6	0.88	15	19.09	12	17.00	18	16.83	21	11.36
REC	18	0.12	2.88	0.31	8	6.40	8	4.39	14	6.67	14	3.19	14	6.67	15	3.12
I-R	4	0.11	2.30	0.37	0	—	0	—	1	15.88	1	7.19	4	6.83	4	2.68
ARR	5	0.33	0.10	1.07	0	—	0	—	1	11.09	3	1.74	1	11.09	3	1.74
LEQ	6	0.10	0.80	0.17	0	—	0	—	0	—	0	—	2	6.32	3	1.11
MON	18	0.05	2.38	(*) 0.15	6	9.62	4	0.25	11	9.77	8	0.97	11	9.77	14	1.43
INJ	11	0.05	1.31	0.15	2	11.38	0		6	55.80	5	1.89	6	55.80	10	1.70
FUN	7	0.05	3.62	0.10	5	4.52	5	0.24	7	5.23	7	0.59	7	5.23	7	0.59
COMP	10	0.26	0.65	19.61	0	—	0	—	3	24.40	6	4.51	6	16.15	9	3.70
Total number:	100				30		23		58		56		69		86	
avg. time:		0.11	2.67	2.24		12.32		1.85		16.53		5.53		14.83		4.41

Table 1. M is the number of verification problems. N is the number of solved problems. Times are in seconds. The timeout is 5 minutes. Sp is CHC Specialization, PP is Predicate Pairing, CP is Constraint Propagation, Eld is ELDARICA. (*) One problem in the category MON timed out.

master branch as of 2016-02-18) using the PDR engine [25] for programs acting on integers and the Duality engine [32] for programs acting on arrays.

The experimental process starts off by applying the Sp strategy. Then, it uses a CHC solver to check the satisfiability of the generated CHCs. If the CHC solver is unable to solve the considered problem, it applies the PP strategy. Finally, if the CHC solver is unable to solve a problem after PP, it applies the CP strategy. In some cases the CP strategy produces a set of CHCs without constrained facts (that is, without clauses of the form $A \leftarrow c$, where c is a constraint), and hence satisfiable, thereby solving the associated problems. In the other cases it applies the CHC solver on the CHCs obtained after constraint propagation.

The experimental process has been performed on a machine equipped with two Intel Xeon E5-2640 2.00GHz CPUs and 64GB of memory, running CentOS 7 (64 bit). A time limit of 5 minutes has been set for executing each step of the experimental process. The verification problems have been executed in parallel using 24 out of the 32 CPU threads. The results are summarized in Table 1.

The first two columns report the names of the categories and the number M of problems in each category, respectively. Columns Sp, PP, and CP report the average time taken for applying those CHC transformation strategies. The Sp and PP strategies terminate before the timeout for all problems, while the CP strategy does not terminate before the timeout for one problem belonging to the MON category. In the remaining columns we report the number of problems solved by the ELDARICA and Z3 solvers after the application of our CHC transformation strategies. We also report the average time taken for each solved problem, which includes the time needed for applying the CHC transformation strategies. In the last two rows we indicate the total number of solved problems and the overall average time.

Columns 1 (Sp + Eld) and 2 (Sp + Z3) report the results for the problems that were solved by applying ELDARICA and Z3, respectively, on the CHCs generated by the Sp strategy. Columns 3 (Sp + PP + Eld) and 4 (Sp + PP + Z3) report the results obtained by applying ELDARICA and Z3, respectively, on the CHCs obtained by the Sp strategy, followed by the PP strategy in the cases where the CHC solvers were unable to produce an answer on the CHCs generated by the Sp strategy. Columns 5 (Sp + PP+ CP + Eld) and 6 (Sp + PP+ CP + Z3) report the results obtained by applying Eldarica and Z3, respectively, on the CHCs produced by the Sp strategy, followed by the PP and CP strategies.

The use of the PP and CP 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 30 (see Column 1) to 69 (Column 5). Similarly for Z3 from 23 (Column 2) to 86 (Column 6). We observe that the application of the PP strategy alone is very effective in increasing the number of solved problems. For instance, it allows Eldarica to solve 28 more problems (see Columns 1 and 3). Also the application of the CP strategy turns out to be very useful for solving additional problems. For instance, for Z3 the CP strategy allows the solution of 30 additional problems (see Columns 4 and 6).

Note that the set of CHCs produced as output by the PP and CP strategies can be larger than the set of CHCs provided as input. In our benchmark we have observed that the increase of size is, on average, about 1.88× for PP and 1.77× for CP (these numbers drop down to 1.77× for PP and 1.16× for CP when we remove from the benchmark 4 examples out of 100 for which the increase of size is very high). However, despite the increase of size, the PP and CP strategies are very effective at improving the efficacy of the considered CHC solvers.

Now, let us compare our experimental results with the ones obtained by RÊVE [20] (the most recent version of the tool is available on-line[6]). For the problems belonging to the ITE and REC categories, we have that if RÊVE succeeds, then also our tool succeeds by using the strategies presented in this paper. As regards the remaining problem categories, an exhaustive comparison with RÊVE is not possible because this tool needs a manual annotation of programs with 'marks', representing synchronization points between programs. Nevertheless, the categories MON, FUN, ARR, and LEQ consist of pairs of programs with a similar control structure (in particular, the programs in MON and FUN are mostly taken from the ITE and REC categories, but with different relational properties), and therefore the approach proposed by Felsing et al. [20] is generally expected to perform well. However, it is worth noting that the problems in I-R and COMP consist of pairs of programs that do *not* exhibit a similar control structure, and therefore the approach of RÊVE with marks cannot be directly applied for solving problems belonging to those categories.

6 Related Work

Several logics and methods have been presented in the literature for reasoning about various relations which may hold between two given programs. Their pur-

[6] `http://formal.iti.kit.edu/improve/reve/index.php`

pose is the formal, possibly automated, validation of program transformations and program analysis techniques.

A Hoare-like axiomatization of relational reasoning for simple while programs has been proposed by Benton [6], which however does not present any technique for automating proofs. In particular, *program equivalence* is one of the relational properties that has been extensively studied (see [5,10,11,20,23,37,38] for some recent work). Indeed, during software development it is often the case that one may want to modify the program text and then prove that its semantics has not been changed (this kind of proofs is sometimes called *regression verification* [20]).

The idea of reducing program equivalence to a standard verification task by using the *cross-product* construction was presented by Zaks and Pnueli [38]. However, this method only applies to structurally equivalent programs. A more refined notion of *program product* has been proposed by Barthe et al. [5] to partially automate general relational reasoning by reducing this problem, similarly to the method proposed by Zaks and Pnueli [38], to a standard program verification problem. The method requires human ingenuity: (i) for generating program products via suitable program refinements and also (ii) for providing suitable invariants to the program verifier. Also the *Differential Assertion Checking* technique proposed by Lahiri et al. [30] 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.

Among the various methods to prove relational properties, the one which is most related to ours is the method proposed by Felsing et al. [20], which presents proof rules for the equivalence of imperative programs that are translated into constrained Horn clauses. The satisfiability of these clauses which entails equivalence, is then checked by state-of-the-art CHC solvers. Although the proof rules are presented for the case of program equivalence, they can be extended to more general relational properties (and, indeed, the tool that implements the method supports a class of specifications comparable to the one presented in this paper). The main difference of our approach with respect to the one of Felsing et al. [20] is that we generate the translation of the relational properties into CHCs from the semantics of the language, and hence we do not need special purpose proof rules that depend on the programming language and the class of properties under consideration. Instead, we use general purpose transformation rules for CHCs. As demonstrated by our experimental evaluation, our approach gives results that are comparable with the ones by Felsing et al. when considering similar examples, but we are able to deal with a larger class of programs. Indeed, besides being more parametric and flexible, an advantage of our approach is that it is able to verify relations between programs that have different structure, because the transformation rules are independent of the syntax of the source programs (unlike the proof rules). For instance, we have shown that we are able to verify relations between while-loops and non-tail recursive functions without a preliminary conversion into tail-recursive form (see Section 2). We are also able to verify relations between programs consisting of the composition of two (or more) while-loops and programs with one while-loop only (see Section 4.3).

The method presented by Felsing et al. is not able to deal with the above two classes of verification problems.

The idea of using program transformations to help the proof of relational properties relating higher-order functional programs has been explored by Asada et al. [3]. The main difference between this approach 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 parametricity 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 by Hawblitzel et al. [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 already mentioned papers by Barthe et al. [5] and Lahiri et al. [30], this approach requires human ingenuity to generate suitable proof obligations, which can then be discharged by automated theorem provers. As regards reusing available verification techniques to prove program equivalence, we want also to mention the paper by Ganty et al. [22], where the authors identify a class of recursive programs for which it is possible to precisely compute summaries. This technique 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.

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 the area of program verification in previous papers [1,12,13,15,16,29,34]. However, the goal of those previous papers was the verification of the (partial and total) correctness of single programs, and not the verification of relations between two programs which has been the objective of our study here.

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 that represent the computations evoked by the programs, and finally, (iii) using state-of-the-art constrained Horn clause solvers to prove satisfiability that enforces the property to be verified.

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.

Acknowledgements

We wish to thank A. Gurfinkel, V. Klebanov, Ph. Rümmer and the participants in the HCVS and VPT workshops at ETAPS 2016 for stimulating conversations.

We also thank the anonymous referees for their very constructive comments. We acknowledge the financial support of INdAM-GNCS (Italy). E. De Angelis, F. Fioravanti, and A. Pettorossi are research associates at IASI-CNR.

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, pp. 124–139. Springer, 2007.
2. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. In *Proc. TACAS* '14, LNCS 8413, pp. 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*, pp. 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, pp. 364–387. Springer, 2006.
5. G. Barthe, J. M. Crespo, and C. Kunz. Relational verification using product programs. In *Proc. FM '11*, LNCS 6664, pp. 200–214. Springer, 2011.
6. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *Proc. POPL '04*, pp. 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, pp. 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, pp. 427–442. Springer, 2006.
10. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *Proc. VMCAI '12*, LNCS 7148, pp. 119–135. Springer, 2012.
11. S. Ciobâcă, D. Lucanu, V. Rusu, and G. Rosu. A language-independent proof system for mutual program equivalence. In *Proc. ICFEM '14*, LNCS 8829, pp. 75–90. Springer, 2014.
12. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95, Part 2:149–175, 2014.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. A rule-based verification strategy for array manipulating programs. *Fundamenta Informaticae*, 140(3-4):329–355, 2015.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. In *Proc. TACAS '14*, LNCS 8413, pp. 568–574. Springer, 2014. `http://www.map.uniroma2.it/VeriMAP`.
15. 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.
16. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Semantics-based generation of verification conditions by program specialization. In *Proc. PPDP '15*, pp. 91–102. ACM, 2015.
17. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pp. 337–340. Springer, 2008.

18. 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.
19. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
20. D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich. Automating regression verification. In *Proc. ASE '14*, pp. 349–360, 2014.
21. J.-C. Filliâtre and A. Paskevich. Why3 - Where programs meet provers. In *Proc. ESOP '13*, LNCS 7792, pp. 125–128. Springer, 2013.
22. P. Ganty, R. Iosif, and F. Konečný. Underapproximation of procedure summaries for integer programs. In *Proc. TACAS '13*, LNCS 7795, pp. 245–259. 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, pp. 282–299. Springer, 2013.
25. K. Hoder and N. Bjørner. Generalized property directed reachability. In *Proc. SAT '12*, pp. 157–171. Springer-Verlag, 2012.
26. 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, pp. 247–251. Springer, 2012.
27. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
28. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
29. B. Kafle and J. P. Gallagher. Constraint specialisation in Horn clause verification. In *Proc. PEPM '15*, pp. 85–90. ACM, 2015.
30. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proc. ESEC/FSE '13*, pp. 345–355. ACM, 2013.
31. Y. V. Matijasevic. Enumerable sets are diophantine. *Doklady Akademii Nauk SSSR* (*in Russian*), 191:279–282, 1970.
32. K. McMillan and A. Rybalchenko. Computing relational fixed points using interpolation. *Technical Report MSR-TR-2013-6*, Microsoft Research, January 2013.
33. G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC '02*, LNCS 2304, pp. 209–265. Springer, 2002.
34. 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, pp. 246–261. Springer, 1998.
35. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
36. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proc. ICLP '84*, pp. 127–138, 1984.
37. 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.
38. A. Zaks and A. Pnueli. CoVaC: Compiler validation by program analysis of the cross-product. In *Proc. FM '08*, LNCS 5014, pp. 35–51. Springer, 2008.