Program Verification using Constraint Handling Rules and Array Constraint Generalizations[⋆]

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

[1] DEC, University 'G. D'Annunzio', Pescara, Italy,
`{emanuele.deangelis,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. The transformation of constraint logic programs (CLP programs) has been shown to be an effective methodology for verifying properties of imperative programs. By following this methodology, we encode the negation of a partial correctness property of an imperative program *prog* as a predicate `incorrect` defined by a CLP program P, and we show that *prog* is correct by transforming P into the empty program through the application of semantics preserving transformation rules. Some of these rules perform replacements of constraints that encode properties of the data structures manipulated by the program *prog*. In this paper we show that Constraint Handling Rules (CHR) are a suitable formalism for representing and applying constraint replacements during the transformation of CLP programs. In particular, we consider programs that manipulate integer arrays and we present a CHR encoding of a constraint replacement strategy based on the theory of arrays. We also propose a novel generalization strategy for constraints on integer arrays that combines the CHR constraint replacement strategy with various generalization operators for linear constraints, such as widening and convex hull. Generalization is controlled by additional constraints that relate the variable identifiers in the imperative program *prog* and the CLP representation of their values. The method presented in this paper has been implemented and we have demonstrated its effectiveness on a set of benchmark programs taken from the literature.

1 Introduction

It has long been recognized that Constraint Logic Programming (CLP) is a formalism that provides very suitable inference mechanisms for the verification of properties of imperative programs. The landmark paper [41] has shown that: (i) the operational semantics of imperative programs can easily be formalized as an interpreter written in CLP, and (ii) by specializing that interpreter with

[⋆] This work has been partially supported by the National Group of Computing Science (GNCS-INDAM).

respect to a given imperative program, say *prog*, one can derive a new CLP program, say *VC*, representing the verification conditions for *prog* in purely logical form. In particular, in the specialized CLP program *VC* there are no references to the imperative constructs of *prog*. Relevant properties of the execution of *prog* (such as its loop invariants) can then be inferred by analyzing the program *VC*.

Many verification methods within the CLP paradigm have been developed. Some methods, directly following the approach presented in [41], are based on *abstract interpretation* [8] and compute an overapproximation of the least model of the CLP program under consideration by a bottom-up evaluation of an abstraction of the program [2, 28, 39]. Other methods use goal directed evaluation of CLP programs combined with other symbolic techniques such as *interpolation* [17, 20, 31, 30]. Some other methods, like the ones presented in [5, 25, 43, 45], combine CLP (also called *constrained Horn clauses* in those papers) with different reasoning techniques developed in the areas of Software Model Checking and *Automated Theorem Proving*, such as CounterExample-Guided Abstraction Refinement (CEGAR) and *Satisfiability Modulo Theory* (SMT).

In this paper we follow the approach based on transformations of CLP programs presented in [12, 13]. We encode the negation of a partial correctness property of an imperative program *prog* as a predicate `incorrect` defined by a CLP program P. Similarly to [41], we generate a CLP program *VC* representing the verification conditions for *prog*, by specializing P with respect to the CLP representation of *prog*. However, at this point the transformation-based method departs from the ones considered above. Indeed, it continues by applying further equivalence preserving transformations to *VC* with the objective of deriving either (i) the empty CLP program, hence proving that `incorrect` does not hold and *prog* is correct, or (ii) a CLP program containing the fact `incorrect`, hence proving that *prog* is incorrect. Due to the undecidability of partial correctness, it may be the case that we derive a CLP program containing one or more clauses of the form `incorrect :- G`, where `G` is a non-empty conjunction, and we are able to conclude neither that *prog* is correct nor that *prog* is incorrect.

Thus, CLP program transformation provides a uniform framework for reasoning about the correctness of imperative programs in which, as we have explained, one can generate the verification conditions and also check their validity. Moreover, that framework is *parametric* with respect to the syntax and the semantics of the programs to be verified, and optimizing transformations considered in the literature [42] can be applied to improve the efficiency of the verification method. Finally, transformations can easily be composed together into a sequence of transformations, so as to derive very sophisticated verification methods. For instance, in [15] it is shown that the *iteration* of program specialization can significantly improve the precision of our program verification method and indeed, by implementing Iterated Specialization the VeriMAP system [14] is competitive with state-of-the-art CLP-based verifiers such as ARMC [43], HSF [25], and TRACER [30].

The main contributions of this paper are the following.

(1) We consider imperative programs that manipulate integers and integer arrays, and we generate verification conditions where read and write operations on arrays are represented as constraints. Then we show that Constraint Handling Rules (CHR) are a suitable formalism for manipulating constraints during the transformation of the CLP verification conditions. In particular, we present CHR rules based on the theory of arrays [7, 23, 37] and we show how they can be combined with *unfold/fold transformation rules* for CLP programs [18] with the objective of proving properties of the given imperative programs.

(2) We propose a powerful transformation strategy that guides the application of both the CHR and the unfold/fold transformation rules. In particular, we design a novel *array constraint generalization strategy* that automatically introduces, during CLP transformation, the new predicate definitions (corresponding to program invariants) required for the verification of the properties of interest. Our generalization strategy combines CHR manipulation of array constraints with the widening and convex hull operators for linear constraints considered in the field of *abstract interpretation* [10]. Generalization is controlled by means of additional constraints that relate the variable identifiers in the given imperative programs and the CLP representations of their values.

(3) Finally, we present an implementation of the method in the VeriMAP system [14], and we demonstrate its effectiveness on a set of benchmark programs taken from the literature.

2 The Transformation-Based Verification Method

In this section we introduce a class of Constraint Logic Programs with constraints on integers and integer arrays, and we show how partial correctness properties of imperative programs can be encoded as programs of that class.

First we need the following definitions. An *atomic integer constraint* is either $\mathtt{p}_1 = \mathtt{p}_2$, or $\mathtt{p}_1 \geq \mathtt{p}_2$, or $\mathtt{p}_1 > \mathtt{p}_2$, where $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and coefficients (sum and multiplication are denoted by `+` and `*`, respectively). An *atomic array constraint* is either $\mathtt{dim(a,n)}$ denoting that the dimension of the array $\mathtt{a}$ is $\mathtt{n}$, or $\mathtt{read(a,i,v)}$ denoting that the $\mathtt{i}$-th element of the array $\mathtt{a}$ is the integer $\mathtt{v}$, or $\mathtt{write(a,i,v,b)}$ denoting that the array $\mathtt{b}$ is equal to the array $\mathtt{a}$, except that its $\mathtt{i}$-th element is $\mathtt{v}$. The `read` and `write` constraints satisfy the following axioms [7, 23], where variables are universally quantified at the front:

(A1) $\mathtt{I = J, read(A,I,U),\ \ read(A,J,V)} \rightarrow \mathtt{U = V}$ (*array congruence*)
(A2) $\mathtt{I = J, write(A,I,U,B), read(B,J,V)} \rightarrow \mathtt{U = V}$ (*read-over-write*:case $=$)
(A3) $\mathtt{I \neq J, write(A,I,U,B), read(B,J,V)} \rightarrow \mathtt{read(A,J,V)}$ (*read-over-write*:case $\neq$)

A *constraint* is either `true`, or an atomic (integer or array) constraint, or a *conjunction* of constraints. An *atom* is a formula of the form $\mathtt{p(t_1,\ldots,t_m)}$, where $\mathtt{p}$ is a predicate symbol not in $\{=, \geq, >, \mathtt{dim}, \mathtt{read}, \mathtt{write}\}$ and $\mathtt{t_1,\ldots,t_m}$ are terms constructed out of variables, constants, and function symbols different from `+` and `*`. A CLP *program* is a finite set of clauses of the form `A :- c, B`, where `A` is an atom, `c` is a constraint, and `B` is a (possibly empty) conjunction of atoms. Given a clause `A :- c, B`, the atom `A` is called the *head*, and `c, B` is called the *body*. We assume that in every clause all integer arguments in its head are distinct variables. A clause `A :- c` is called a *constrained fact*. If `c` is `true`, then it is omitted and the constrained fact is called a *fact*. A CLP program is said to be *linear* if all its clauses are of the form `A :- c, B`, where `B` consists of at most one atom.

An $\mathcal{A}$-*interpretation* I is a set D, together with a function f in $D^n \rightarrow D$ for each function symbol $\mathtt{f}$ of arity n, and a relation p on D^n for each predicate symbol $\mathtt{p}$ of arity n, such that: (i) the set D is the Herbrand universe [36] constructed out of the set $\mathbb{Z}$ of the integers, the constants, and the function symbols different from + and (ii) I assigns to the symbols $+, *, =, \geq, >$ the usual meaning in $\mathbb{Z}$, (iii) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$, for all integers $\mathtt{d}$, $\mathtt{dim}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{d})$ is true in I iff $\mathtt{d} = \mathtt{n}$, (iv) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ and $\mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1}$ of integers, for all integers $\mathtt{i}$ and $\mathtt{v}$, $\mathtt{read}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v})$ is true in I iff $0 \leq \mathtt{i} \leq \mathtt{n}-1$ and $\mathtt{v} = \mathtt{a}_\mathtt{i}$, and $\mathtt{write}(\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}, \mathtt{i}, \mathtt{v}, \mathtt{b}_0 \ldots \mathtt{b}_{\mathtt{m}-1})$ is true in I iff $0 \leq \mathtt{i} \leq \mathtt{n}-1$, $\mathtt{n} = \mathtt{m}$, $\mathtt{b}_\mathtt{i} = \mathtt{v}$, and for $\mathtt{j} = 0, \ldots, \mathtt{n}-1$, if $\mathtt{j} \neq \mathtt{i}$ then $\mathtt{a}_\mathtt{j} = \mathtt{b}_\mathtt{j}$, (v) I is an Herbrand interpretation [36] for function and predicate symbols different from $+, *, =, \geq, >$, `dim`, `read`, and `write`.

We can identify an $\mathcal{A}$-interpretation I with the set of all ground atoms that are true in I, and hence $\mathcal{A}$-interpretations are partially ordered by set inclusion. A constraint `c` is said to be *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where in general, for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. We say that I is an $\mathcal{A}$-*model* of φ if φ is true in I. We write $\mathcal{A} \models \varphi$ if every $\mathcal{A}$-interpretation is an $\mathcal{A}$-model of φ. In particular, every $\mathcal{A}$-interpretation is an $\mathcal{A}$-model of Axioms (A1)–(A3). A constraint `c` *entails* a constraint `d`, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. By *vars*$(\varphi)$ we denote the free variables of φ. The semantics of a CLP program P is *the least* $\mathcal{A}$-*model* of P, denoted $M(P)$ and constructed as usual for CLP programs [29].

We consider imperative programs with integer and array variables. Every program has a single `halt` command whose execution causes the program to terminate. The semantics of programs is defined in terms of a *transition relation*, denoted $\Longrightarrow$, between *configurations*. A configuration is a pair $\langle\!\langle c, \delta \rangle\!\rangle$ of a *labeled command* c and an *environment* δ that maps: (i) every integer variable identifier x to its value v, and (ii) every integer array identifier a to a *finite* sequence $\mathtt{a}_0 \ldots \mathtt{a}_{n-1}$ of integers, where $\mathtt{n}$ is the dimension of the array a. The transition relation specifies the 'small step' operational semantics and its definition is similar to that in [44] and is omitted. An environment δ is said to satisfy a formula $\varphi(z_1, \ldots, z_r)$ iff $\varphi(\delta(z_1), \ldots, \delta(z_r))$ holds.

Given two formulas φ_{init} and φ_{error} that are disjunctions of constraints with free variables $z_1, \ldots, z_r$, we say that program *prog* is *incorrect* with respect to these formulas iff there exist two environments δ_{init} and δ_h such that: (i) δ_{init} satisfies φ_{init}, (ii) $\langle\!\langle \ell_0 \!:\! c_0, \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h \!:\! \mathtt{halt}, \delta_h \rangle\!\rangle$, and (iii) δ_h satisfies φ_{error}, where $\ell_0 \!:\! c_0$ is the first labeled command of *prog* and $\ell_h : \mathtt{halt}$ is the unique `halt` command of *prog*. A program is said to be *correct* if it is not incorrect. (In [11] the reader may find an extension of these definitions where φ_{init} and φ_{error} are predicates defined by any CLP program.) Our notion of correctness is equivalent to the Hoare notion of *partial correctness* specified by the triple $\{\varphi_{init}\}$ *prog* $\{\neg\, \varphi_{error}\}$.

We translate the problem of checking whether or not the program *prog* is *incorrect* into the problem of checking whether or not the atom `incorrect` is a consequence of the following CLP program T:

```
incorrect :- errorConf(X), reach(X).
reach(Y) :- tr(X,Y), reach(X).
reach(Y) :- initConf(Y).
```

where `initConf(X)`, `errorConf(X)`, and `tr(X,Y)` are defined by CLP clauses so that the following conditions hold. For all configurations `X` and `Y`, (i) `initConf(X)` holds iff `X` is an *initial configuration*, that is, a configuration of the form $\langle\!\langle \ell_0 : c_0, \delta_{init} \rangle\!\rangle$ and δ_{init} satisfies φ_{init}, (ii) `errorConf(X)` holds iff `X` is an *error configuration*, that is, a configuration of the form $\langle\!\langle \ell_h{:}\mathtt{halt}, \delta_h \rangle\!\rangle$ and δ_h satisfies φ_{error}, and (iii) `tr(X,Y)` holds iff `X` $\Longrightarrow$ `Y` holds.
`reach(Y)` holds iff the configuration `Y` can be reached from a configuration `X` whose environment satisfies φ_{init}. Program *prog* is correct with respect to φ_{init} and φ_{error} iff `incorrect`$\notin M(T)$.

Our verification method applies unfold/fold rules to the initial program T and consists of following two steps [13]. (i) *VCGen*: the Generation of the Verification Conditions, and (ii) *VCTransf*: the Satisfiability Checking of the Verification Conditions. The soundness of our method follows from the fact that for each program U obtained from T by applying the unfold/fold rules, `incorrect` $\in M(T)$ iff `incorrect`$\in M(U)$.

VCGen performs a *specialization* of program T with respect to the given `tr` (which depends on *prog*), `initConf`, and `errorConf` predicates, thereby deriving a new program $T1$, whose clauses are said to be the *verification conditions* for *prog*, such that `tr` does not occur in $T1$ (for this reason this step is also called *the removal of the interpreter*). During this specialization step all occurrences of the `dim` predicate are replaced by suitable constraints on the indexes of the arrays. We say that verification conditions are *satisfiable* iff `incorrect` $\notin M(T1)$, and thus their satisfiability guarantees that *prog* is correct with respect to φ_{init} and φ_{error}. *VCTransf*, which will be described in detail in Section 3, checks the satisfiability of the verification conditions generated at the end of *VCGen*.

Before starting the specialization, *VCGen* adds to the initial program T some additional constraints that are needed for controlling the generalization strategy described in Section 3.3. These constraints use the predicate `val` that relates some of the variable identifiers occurring in the imperative program *prog* and the CLP representation of their values. The meaning of the `val` constraints is as follows: for every variable identifier i of the program *prog*, for every value `I`, the constraint `val(i,I)` (where `i` is a constant uniquely associated with i) holds iff there exists a configuration whose environment δ maps i to `I`. These `val` constraints will be used by our generalization strategy to distinguish among different `read` constraints, thereby making the strategy more effective as confirmed by the experimental results reported in Section 4. For instance, the constraint '`val(i,I),val(j,J),read(A,I,U),read(A,J,V)`' expresses the property that the first `read` gets the array element at index i and the second `read` gets the array element at index j, while without the `val` constraints, '`read(A, I, U), read(A, J, V)`' does not express this property.

Now, let us see our verification method in action on a simple example. Let us consider the following program that, given the array $a[0\,..\,(n-1)]$ and any $i \in \{0,\ldots,n-1\}$ places in $a[n-i-1]$ the maximum value of the leftmost portion $a[0\,..\,(n-i-1)]$ by iteratively swapping adjacent elements.

bubblesort-inner: `for` $(j=0;\;\; j<n-i-1;\;\; j{+}{+})$ {
    `if` $(a[j] > a[j{+}1])$ $\{tmp = a[j];\; a[j] = a[j{+}1];\; a[j{+}1] = tmp;\}$ }

Let us also consider the two properties $\varphi_{init}(i,n,a) \equiv 0 \leq i < n \;\wedge\; dim(a,n)$ and $\varphi_{error}(i,j,n,a) \equiv \exists k \exists x \exists y\, 0 \leq i < n \wedge 0 \leq k < j \wedge j = n - i - 1 \wedge read(a,k,x) \wedge read(a,j,y) \wedge x > y$.

These two properties are expressed in CLP as follows:

```
phiInit(I, N, A) :- 0≤I, I<N, dim(A, N).
phiError(I, J, N, A) :- 0≤I, I<N, 0≤K, K<J, J=N−I−1, X>Y, read(A,K,X), read(A,J,Y),
                        val(k, K), val(j, J).
```

Note the two `val` constraints that relate the index variables k and j to their values `K` and `J`, respectively. At the end of *VCGen* we get the following CLP program $T1$ that expresses the verification conditions for the program *bubblesort-inner*:

1. `incorrect :- 0≤I, 0≤K, K≤J,` $\underline{\texttt{J=N-I-1}}$`, X>Y,`
   `read(A, K, X), read(A, J, Y), val(k, K), val(j, J), new1(I, J, N, A, Tmp, K).`
2. `new1(I, J1, N, A2, W, K) :- J1=1+J, J<N−I−1, J≥0, J<N−1,` $\underline{\texttt{X>Y}}$`,`
   `read(A, J, X), read(A, J1, Y), read(A, J, W), read(A, J1, Z), write(A, J, Z, A1),`
   `write(A1, J1, W, A2), val(j, J1), val(j, J), val(k, K), new1(I, J, N, A, Tmp, K).`
3. `new1(I, J1, N, A, Tmp, K) :- J1=J+1, J<N−I−1, J≥0, J<N−1,` $\underline{\texttt{X}\leq\texttt{Y}}$`,`
   `read(A, J, X), read(A, J1, Y), val(j, J1), val(j, J), val(k, K), new1(I, J, N, A, Tmp, K).`
4. `new1(I, J, N, A, Tmp, K) :- 0≤I, I<N,` $\underline{\texttt{J=0}}$`, val(j, J), val(k, K).`

where `new1` is a new predicate symbol introduced during program specialization by *VCGen*. The definition of the predicate `new1` is associated with the `for`-loop of the *bubblesort-inner* program and consists of clauses 2–4 that represent the execution of the `for` statement. In particular, we have that (see the underlined constraints): (i) clauses 1 and 4 represent the exit and the entry of the `for`-loop, respectively, and (ii) clauses 2 and 3 represent the execution of the conditional in the $a[j] > a[j+1]$ case and in the $a[j] \leq a[j+1]$ case, respectively.

3 A Transformation Strategy for Verification

The *VCTransf* step of our verification method transforms the CLP program $T1$ derived at the end of *VCGen* to a program $T2$ such that `incorrect` $\in M(T1)$ iff `incorrect` $\in M(T2)$. This transformation makes use of *transformation rules* that preserve the least $\mathcal{A}$-model semantics of CLP programs. In particular, we apply the following rules, that are collectively called unfold/fold rules: *unfolding*, *constraint replacement*, *clause removal*, *definition*, and *folding*. These rules are an adaptation to CLP programs of the unfold/fold rules for general CLP programs (see, for instance, [18]).

VCTransf applies the unfold/fold rules according to a strategy that performs the propagation of the constraints of the error property `phiError` in a backward way from the error configuration towards the initial configuration, so as to derive a program $T2$ where the predicate `incorrect` is defined by either (i) the fact `incorrect` (in which case the imperative program *prog* is incorrect), or (ii) the empty set of clauses (in which case *prog* is correct). In the case where neither (i) nor (ii) holds, that is, in program $T2$ the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact `incorrect`, we cannot conclude anything about the correctness of *prog*. However, similarly to what has been proposed in [12], in this case we can perform again *VCTransf* by propagating the initial property `phiInit`, and continue alternating the propagation of the error and initial properties in the hope of deriving a program where either (i) or (ii) holds. Obviously, due to the undecidability of program correctness, it may be the case that this process does not terminate.

3.1 The Transformation Strategy

VCTransf is performed by applying the unfold/fold transformation rules according to the *Transform* strategy shown in Figure 1. Let us briefly describe the various rules used by the *Transform* strategy.

• The Unfolding rule performs one step of backward propagation of the error property `phiError`.

• The Constraint Replacement rule infers new constraints on the variables of the single atom that occurs in the body of each clause obtained by Unfolding. Constraint Replacement makes use of a function *Repl* that, given a clause C of the form $\mathtt{H} \mathbin{:\!-} \mathtt{c}_0, \mathtt{B}$, returns a set $\{\mathtt{H} \mathbin{:\!-} \mathtt{c}_1, \mathtt{B}, \ldots, \mathtt{H} \mathbin{:\!-} \mathtt{c}_\mathtt{n}, \mathtt{B}\}$ of clauses (with $\mathtt{n} \geq 0$), where $\mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{n}$ are constraints such that $\mathcal{A} \models \forall((\exists \mathtt{X}_0\, \mathtt{c}_0) \leftrightarrow (\exists \mathtt{X}_1\, \mathtt{c}_1 \vee \ldots \vee \exists \mathtt{X}_\mathtt{n}\, \mathtt{c}_\mathtt{n}))$ holds, and for $i = 0, \ldots, \mathtt{n}$, we have that $\mathtt{X}_\mathtt{i} = \mathit{vars}(\mathtt{c}_\mathtt{i}) - \mathit{vars}(\mathtt{H}, \mathtt{B})$. In particular, if $\mathtt{c}_0$ is unsatisfiable, then $\mathtt{n} = 0$ and clause C is removed. The function *Repl* is implemented by a CHR program as described in Section 3.2.

• The rules of Removal of Useless Clauses and Removal of Subsumed Clauses remove clauses that do not contribute to the least model of the CLP program at hand.

• The Definition rule introduces new predicate definitions by suitable generalizations of the constraints. Generalization is performed by using a function *Gen* such that, for any given clause E of the form `H :- e(V,X), p(X)` and set *Defs* of predicate definitions, *Gen*(E, *Defs*) is a clause of the form `newq(X) :- gen(X), p(X)`, where: (i) `newq` is a new predicate symbol, and (ii) `gen(X)` is a constraint such that $\mathtt{e(V,X)} \sqsubseteq \mathtt{gen(X)}$.

• The Folding rule replaces the clause `H:-e(V,X), p(X)` by the clause `H:-e(V,X), newq(X)`.

Note that the input program $T1$ of the *Transform* strategy is a *linear* CLP program. Indeed, during *VCGen* the atoms different from `reach` are unfolded and hence a linear program is generated.

The new predicates introduced by the Definition rule can be understood as *over-approximations* of the sets of configurations that are backward-reachable from the error configuration. Note, however, that the folding rule preserves equivalence, as `e(V,X), p(X)` is equivalent to `e(V,X), newq(X)`. In Section 3.3 we present a generalization function that guarantees the termination of *Transform* and, at the same time, allows us to prove the correctness of non-trivial programs.

```
Input: A linear CLP program $T1$.
Output: Program $T2$ such that incorrect $\in M(T1)$ iff incorrect $\in M(T2)$.

INITIALIZATION:
Let InDefs be the set of all clauses of $T1$ whose head is the atom incorrect;
$T2 := \emptyset$;    Defs := InDefs;
while in InDefs there is a clause $C$ of the form H :- c,A  do
    UNFOLDING: Let $\{K_i$ :- $c_i, B_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart)
        clauses of $T1$ such that, for $i = 1, \ldots, m$, A is unifiable with $K_i$ via the most
        general unifier $\vartheta_i$.
        Then TransfC := $\{($H :- c,$c_i, B_i)\,\vartheta_i \mid i = 1, \ldots, m\}$;
    CONSTRAINT REPLACEMENT: TransfC := $\cup_{D \in TransfC} Repl(D)$;
    REMOVAL OF SUBSUMED CLAUSES: Remove from TransfC every clause H :- d,B
        such that there exists a distinct clause H :- e in TransfC with d $\sqsubseteq$ e;
    DEFINITION & FOLDING:
    while in TransfC there is a clause $E$ of the form H :- e(V,X), p(X), where e(V,X)
    is a constraint and p is a predicate defined in $T1$ do
      if in Defs there is a clause $D$ of the form newp(X) :- c(X), p(X), where c(X)
        is a constraint such that e(V,X) $\sqsubseteq$ c(X)
      then TransfC := (TransfC $-$ $\{E\}$) $\cup$ {H :- e(V,X), newp(X)};
      else let Gen($E$, Defs) be newq(X) :- gen(X), p(X);
          Defs := Defs $\cup$ {Gen($E$, Defs)};
          InDefs := (InDefs $-$ $\{C\}$) $\cup$ {Gen($E$, Defs)};
          TransfC := (TransfC $-$ $\{E\}$) $\cup$ {H :- e(V,X), newq(X)}
    end-while;
    $T2 := T2 \cup$ TransfC
end-while;
REMOVAL OF USELESS CLAUSES:
Remove from $T2$ all clauses with head predicate p, if in $T2$ there is no constrained fact
q(...) :- c where q is either p or a predicate on which p depends.
```

Fig. 1. The *Transform* strategy.

We assume that the set *Defs* is structured as a tree of clauses where, with reference to Figure 1, clause C is said to be the *parent* of clause *Gen*$(E, Defs)$, and the *ancestor* relation is defined as the reflexive, transitive closure of the parent relation.

3.2 Constraint Replacement via CHR

In this section we show how Constraint Handling Rules with disjunction can be used to realize in a very natural way the constraint rewritings based on Axioms (A1)–(A3) for array operations, which allow us to apply the Constraint Replacement rule during the *Transform* strategy.

CHR is a committed-choice language based on rewriting rules. It was specifically designed for building custom constraint solvers [22]. A CHR program consists of a set of guarded rules that rewrite multisets of constraints. Constraint predicates are of two different kinds: (i) *built-in constraints*, whose entailment is checked by using a domain-specific constraint solver, and (ii) *user-defined constraints*, which are rewritten as specified by the CHR program. We assume that the set of built-in constraints contains the constraints `true`, `false`, and syntactic equalities. Built-in constraints and user-defined constraints are closed under conjunction. A *constraint goal* is either a (built-in or user-defined) constraint, or a conjunction of constraint goals, or a disjunction of constraint goals.

CHR rules are of the form: $\mathtt{r} \ @ \ \mathtt{H}_1 \setminus \mathtt{H}_2 \Leftrightarrow \mathtt{G} \mid \mathtt{B}$, where the @ symbol separates the optional rule identifier `r` from the rest of the rule, the user-defined constraints $\mathtt{H}_1$ and $\mathtt{H}_2$ are the *kept head* and the *removed head*, respectively, the built-in constraint `G` is the *guard*, and `B` is a constraint goal. Either $\mathtt{H}_1$ or $\mathtt{H}_2$ is a non-empty conjunction. If $\mathtt{H}_2$ is empty then the rule is called a *propagation rule* and can be written as follows: $\mathtt{H}_1 \Rightarrow \mathtt{G} \mid \mathtt{B}$. The logical meaning of the CHR rule $\mathtt{H}_1 \setminus \mathtt{H}_2 \Leftrightarrow \mathtt{G} \mid \mathtt{B}$ is the guarded equivalence $\forall(\mathtt{G} \rightarrow ((\mathtt{H}_1 \wedge \mathtt{H}_2) \leftrightarrow (\mathtt{H}_1 \wedge \exists \mathtt{Y}\,\mathtt{B})))$, where `Y` is the set of variables occurring in `B` and not in the rest of the rule.

The operational semantics of CHR is formally defined in terms of a transition relation between CHR *states* as described in [1]. A CHR state is a triple $\langle \mathtt{g}, \mathtt{u}, \mathtt{b} \rangle$, where `g` is a constraint goal, `u` is a user-defined constraint and `b` is a built-in constraint. An *initial state* is a state of the form $\langle \mathtt{g}, \mathtt{true}, \mathtt{true} \rangle$. Starting from an initial state, constraints are rewritten as long as possible by applying CHR rules. A *final state* is a state from which no transition is applicable. A final state is *failed* if it is of the form $\langle \mathtt{g}, \mathtt{u}, \mathtt{false} \rangle$. Note that, since constraint goals may contain disjunctions, the transition relation is nondeterministic, and thus it generates a tree of computations whose leaves correspond to the final states. A *terminating* CHR program is one for which there is no infinite sequence of transitions, that is, the tree of computations is finite.

The CHR program `Arr` used for constraint replacement in the *Transform* strategy consists of the following rules:

```
ac  @ read(A1, I, X)\read(A2, J, Y) ⇔ A1 == A2, I = J     | X = Y.
cac @ read(A1, I, X), read(A2, J, Y)  ⇒ A1 == A2, X <> Y | I <> J.
row @ write(A1, I, X, A2)\read(A3, J, Y) ⇔ A2 == A3 | (I = J, X = Y); (I <> J, read(A1,J,Y)).
```

These rules encode the axioms (A1)–(A3) presented in Section 2. Rules `ac` and `cac` encode the array congruence axiom (A1) and its contrapositive version, respectively, and rule `row` encodes the two so-called read-over-write axioms (A2) and (A3). The symbol '`==`' denotes syntactic equality, while '`=`' and '`<>`' denote integer equality and inequality, respectively. Note that we use the semicolon '`;`' for denoting disjunction in the right-hand side of the rule `row`.

If we adopt an operational semantics that prevents trivial non-termination cases by applying a propagation rule at most once to the same constraints [1], then it can be shown that the CHR program `Arr` terminates for all constraint goals generated during the application of our transformation strategy. Indeed, the only rule that may lead to a non-terminating behavior is `row`. By using this rule, a constraint containing

(g1) $\underline{\texttt{write(U,I,X,V)}}$, `write(V,I,H,U)`, $\underline{\texttt{read(V,J,Y)}}$

could be rewritten as a constraint containing

(g2) $\underline{\texttt{write(U,I,X,V)}}$, `write(V,I,H,U)`, $\underline{\texttt{read(U,J,Y)}}$

and then, by interchanging the roles of the two `write` constraints in the application of the `row` rule, a constraint containing (g2) could be rewritten to a constraint containing (g1), thereby giving rise to an infinite branch in the tree of computation. However, it can be shown that a constraint goal of the form (g1) cannot be generated by the UNFOLDING rule during the application of the *Transform* strategy. Informally, in every clause, the constraints can be ordered from left to right following the order of execution of the corresponding `read` and `write` operations, and hence a variable V occurring in a constraint of the form write(U, I, X, V), does not occur to the left of that constraint. This argument is formalized by considering the transitive closure $\prec^+$ of the following relation between the variables of a clause: $\mathtt{U} \prec \mathtt{V}$ iff the constraint write(U, I, X, V) occurs in the clause. It can be shown that in every clause derived by the UNFOLDING rule during the application of the *Transform* strategy, $\prec^+$ is irreflexive. Thus, the termination of `Arr` follows from the fact that an application of the `row` rule will replace a constraint of the form `read(V,J,Y)` by a constraint of the form `read(U,J,Y)` with $\mathtt{U} \prec \mathtt{V}$.

Given a clause D of the form H :- d, B, derived by the UNFOLDING rule, let $\{\langle \mathtt{g}_1, \mathtt{u}_1, \mathtt{b}_1\rangle, \ldots, \langle \mathtt{g}_n, \mathtt{u}_n, \mathtt{b}_n\rangle\}$ be the set of all non-failed final states computed from the initial state $\langle \mathtt{d}, \mathtt{true}, \mathtt{true}\rangle$. Let $\mathtt{d}_i$ be the conjunction $\langle \mathtt{g}_i, \mathtt{u}_i, \mathtt{b}_i\rangle$. We assume that, for $\mathtt{i} = 1, \ldots, \mathtt{n}$, the variables occurring in $\mathtt{d}_i$ and not in d are fresh, and thus they occur neither in H nor in B. By the soundness of CHR we have that $\mathcal{A} \models \forall(\mathtt{d} \leftrightarrow (\exists \mathtt{X}_1\, \mathtt{d}_1 \vee \ldots \vee \exists \mathtt{X}_n\, \mathtt{d}_n))$ where, for $\mathtt{i} = 1, \ldots, \mathtt{n}$, $\mathtt{X}_i = \mathit{vars}(\mathtt{d}_i) - \mathit{vars}(\mathtt{d})$. Thus, the applicability conditions of the CONSTRAINT REPLACEMENT rule are satisfied, and in the *Transform* strategy we define *Repl*(D) to be $\{\mathtt{H} \mathrel{:\!-} \mathtt{d}_1, \mathtt{B}, \ldots, \mathtt{H} \mathrel{:\!-} \mathtt{d}_n, \mathtt{B}\}$.

To see how the CHR program `Arr` works, let us consider again the *bubblesort-inner* example of Section 3. By applying the UNFOLDING rule to clause 1 the *Transform* strategy derives a set of clauses including the following one:

```
new2(I,J1,N,A2,W,K) :- J1=1+J, J<N−I−1, K≤J, Z<W, I≥0, K≥0, J≥N−I−3, X>Y,
    write(A,J,Z,A1), write(A1,J1,W,A2), read(A,J,W), read(A,J1,Z),
    read(A2,K,X), read(A2,J1,Y), val(j,J1), val(k,K), val(j,J), new1(I,J,N,A,Tmp,K).
```

The CHR program `Arr` rewrites the constraint occurring in the above clauses and the CONSTRAINT REPLACEMENT rule derives the following clause:

```
new2(I,J1,N,A2,W,K) :- J1=1+J, J<N−I−1, K≤J, Z<W, I≥0, K≥0, J≥N−I−3, X>Y,
    write(A,J,Z,A1), write(A1,J1,W,A2), read(A,J,Y), read(A,J1,Z),
    read(A,K,X), Y=W, J>K, J1>K, val(j,J1), val(k,K), val(j,J), new1(I,J,N,A,Tmp,K).
```

where (i) by `row`, the constraint `read(A2, J1, Y)` has been replaced by the equality constraint `Y = W` (ii) by `row`, in the constraint `read(A2, K, X)`, the variable `A2`, denoting the array a after the `write` operation, has been replaced by the variable `A`, denoting the array a before the `write` operation, and (iii) the constraint '`J > K, J1 > K`' has been added by the built-in solver on linear constraints.

3.3 Generalization Strategy

The most critical step of the *Transform* strategy is the introduction of new predicates during Definition & Folding. Indeed, it should guaranteed that a finite number of new predicates is introduced, to avoid the non-termination of *Transform*. For this reason, as usual in many program transformation techniques [19], we collect in the set *Defs* all predicate definitions introduced by the strategy, and before introducing a new predicate definition D, we match it against the ones already in *Defs*. If D is 'similar' to a definition A in *Defs* (formalized via the *embedding* relation defined below), then the function *Gen* introduces a new definition which is a generalization of A and D, instead of D. The function *Gen* defined in this section, makes use of operators for generalizing array constraints that ensure that no infinite number of distinct generalizations can be obtained, and hence a finite number of new predicates is introduced during the *Transform* strategy. The embedding relation and the generalization strategy take into consideration the `val` constraints between the integer CLP variables occurring in `read` constraints and the identifiers of the imperative program with which they are associated. By doing so we will be able to identify similarities between definitions that go beyond syntactic variance, hence improving the level of precision of the verification technique.

In the following we will denote constraints as conjunctions of the form `i, r, w, v`, where `i` is an integer constraint, and `r`, `w`, and `v` are conjunctions of `read`, `write`, and `val` constraints, respectively. We assume that all integer variables in `read` constraints are distinct and do not occur in any (non constraint) atom of the clause at hand (this condition can always be satisfied by adding some integer equalities).

Given a clause D of the form `H :- i, r, w, v, B`, for every integer variable `I` occurring in a `read` atom in `r` we compute the set *ids*(`I`) of identifiers `id` such that an atom `val(id, J)` occurs in `v` and the constraint `I = J` is entailed by `i`. We define the *clause identifier set* of D, denoted *ids*(D), as the set of pairs (*ids*(`I`), *ids*(`U`)) such that a constraint of the form `read(A, I, U)` occurs in `r`. For example, if the constraint occurring in the body of clause D is

`M = 0, N > M, V = 0, read(A, M, U), read(A, N, V), val(m, M), val(n, N), val(v, V)`
then we have that *ids*$(D) = \{(\{\mathtt{m}, \mathtt{v}\}, \{\}), (\{\mathtt{n}\}, \{\mathtt{m}, \mathtt{v}\})\}$.

Given two clause identifier sets R_1 and R_2, we say that R_1 is *embedded* into R_2 via the set relation *rel* iff for each pair (I_1, U_1) in R_1 there exists a pair (I_2, U_2) in R_2 such that (i) $rel(I_1, I_2)$ and $rel(U_1, U_2)$ hold and (ii) $R_1 - \{(I_1, U_1)\}$ is embedded into $R_2 - \{(I_2, U_2)\}$ via *rel*. In our experiments we have considered two embedding relations based on the following definitions of $rel(s_1, s_2)$: (1) $s_1 \subseteq s_2$ (subset relation), and (2) $s_1 \Cap s_2$ defined as $(s_1 = s_2 = \emptyset) \vee (s_1 \cap s_2 \neq \emptyset)$.

We say that a clause D_1 is embedded into a clause D_2 via the relation *rel* iff $ids(D_1)$ is embedded in $ids(D_2)$ via *rel*.

Given a clause E of the form `H :- e(V,X), p(X)` and a set *Defs* of definitions, the generalization function *Gen* computes a definition `newq(X) :- gen(X), p(X)`, where `newq` is a new predicate symbol and `gen(X)` is a constraint such that `e(V,X)` $\sqsubseteq$ `gen(X)`, which is constructed as follows. Let $\mathtt{e(V,X)}$ be of the form $\mathtt{i,r,w,v}$ and let $\mathtt{newq(X) :\!- i_X, r_X, v_X,\ p(X)}$ be the *candidate definition* clause for E, where: (i) $\mathtt{r_X}$ is the conjunction of the $\mathtt{read(A,I,V)}$ constraints in `r` such that `A` occurs in `X` and, for some $\mathtt{val(j,J)}$ in `v` we have that `J` occurs in `X` and either $\mathtt{I = J}$ or $\mathtt{V = J}$ is entailed by `i`, (ii) $\mathtt{i_X}$ is the constraint obtained from `i` by *projecting* away the variables not occurring in `X` or $\mathtt{r_X}$, and (iii) $\mathtt{v_X}$ is the conjunction of the $\mathtt{val(j,J)}$ constraints in `v` such that `J` occurs in `X`.

Suppose that clause E has been derived from clause C at the end of the Removal of Subsumed Clauses step. $Gen(E, Defs)$ is defined as follows.

If in *Defs* there is an ancestor A of C of the form $\mathtt{H_0}$:- $\mathtt{i_0, r_0, v_0, p(X)}$, such that $\mathtt{r_0}$ is a subconjunction of $\mathtt{r_X}$, and A is embedded into $\mathtt{newq(X) :\!- i_X, r_X, v_X,\ p(X)}$,

Then let $\mathtt{i_1}$ be the constraint obtained from $\mathtt{i_X}$ by projecting away the variables not occurring in `X` or $\mathtt{r_0}$; compute a generalization `g` of the constraints $\mathtt{i_1}$ and $\mathtt{i_0}$ such that $\mathtt{i_1} \sqsubseteq \mathtt{g}$, by using a *generalization operator* for linear constraints. Define the constraint $\mathtt{gen(X)}$ as $\mathtt{g, r_0, v_0}$;

Else define the constraint $\mathtt{gen(X)}$ as $\mathtt{i_X, r_X, v_X}$.

For the projection and generalization operations we apply the usual operators for linear constraints on the reals (and in particular the *widening* and *convex hull* generalization operators defined in [10, 19, 40]). These operators are correct because they guarantee that $\mathtt{i} \sqsubseteq \mathtt{g}$.

To see an example of application of the generalization strategy let us consider the clause that was derived in Section 3.2 by applying the Constraint Replacement rule. The candidate definition for that clause is:

```
new4(I,J,N,A,Tmp,K) :- J<N−I−1, I≥0, K≥0, J≥N−I−3, X>W, J>K,
          read(A,J,W), read(A,K,X), val(k,K), val(j,J), new1(I,J,N,A,Tmp,K).
```

and *Defs* contains the following ancestor definition:

```
new2(I,J,N,A,Tmp,K) :- J<N−I−1, I≥0, K≥0, J≥N−I−2, X>W, J>K,
          read(A,J,W), read(A,K,X), val(k,K), val(j,J), new1(I,J,N,A,Tmp,K).
```

Since the ancestor definition is embedded into the candidate definition via $\subseteq$ or $\Subset$ (indeed, the two clauses have the same clause identifier set $\{(\{\mathtt{j}\},\{\}),(\{\mathtt{k}\},\{\})\}$), we obtain a generalization of the candidate definition by applying the widening operator between the linear constraints, hence dropping the constraint $\mathtt{J \geq N-I-2}$ of the ancestor definition, and we introduce the following generalized definition:

```
new4(I,J,N,A,Tmp,K) :- J<N−I−1, I≥0, K≥0, X>W, J>K,
          read(A,J,W), read(A,K,X), val(k,K), val(j,J), new1(I,J,N,A,Tmp,K).
```

The correctness of the *Transform* strategy with respect to the least $\mathcal{A}$-model semantics follows from the correctness results for the unfold/fold rules proved in [18].

The termination of the *Transform* strategy is based on the following facts: (i) Constraint satisfiability and entailment are checked by a terminating solver (note that completeness is not necessary for the termination of *Transform*). (ii) The CHR program `Arr` implementing Constraint Replacement terminates. (iii) The set of new clauses that, during the execution of the *Transform* strategy, can be introduced by Definition & Folding steps is finite. Indeed, by construction, they are all of the form `H :- i, r, v, p(X)`, where: (1) `X` is a tuple of variables, (2) `i` is an integer constraint, (3) `r` is a conjunction of array constraints of the form `read(A, I, V)`, where `A` is a variable in `X` and the variables `I` and `V` occur in `i` only, (4) the set of identifiers of the imperative program is finite, and hence the embedding relation is a *thin well-quasi ordering* [19] (this property guarantees that generalization is eventually triggered, and that a definition can be generalized a finite number of times only), (5) the cardinality of `r` is bounded, because if in *Defs* there exists a clause A of the form $\mathtt{H_0} \text{ :- } \mathtt{i_0}, \mathtt{r_0}, \mathtt{v_X}, \mathtt{p(X)}$, then generalization does not introduce a descendant definition clause D of the form $\mathtt{newp(X)} \text{ :- } \mathtt{i_X}, \mathtt{r_0}, \mathtt{r_1}, \mathtt{v_X}, \mathtt{p(X)}$ such that A is embedded into D, (6) we assume that the generalization operator on linear constraints has the following *finiteness* property: only finite chains of generalizations of any given constraint can be generated by applying the operator. The already mentioned generalization operators presented in [10, 19, 40] satisfy this finiteness property. Thus, we have the following result.

Theorem 1. (i) *The Transform strategy terminates.* (ii) *Let program $T2$ be the output of Transform applied to the input program $T1$. Then,* `incorrect` $\in M(T1)$ *iff* `incorrect` $\in M(T2)$.

Let us now conclude our *bubblesort-inner* example. After a few iterations, the outermost while-loop of the *Transform* strategy terminates and produces the following set $T2$ of clauses (which we list as they have been automatically generated):

```
incorrect :- A = −1+B−C, D = −1+B−C, E−F ≤ −1, G ≥ 0, C ≥ 0, B−G−C ≥ 2,
    read(H,D,E), read(H,G,F), val(j,A), val(k,G), new1(C,A,B,H,I,G).
new1(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, A = −2+C−G, B = 1+G, I = 1+G, H = 1+G,
    J = 1+G, K = 1+G, F−G ≤ 0, L−E ≤ −1, F ≥ 0, C−G ≥ 2, M−E ≥ 1,
    read(N,F,M), read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
    val(j,G), val(k,F), val(j,B), new2(A,G,C,N,P,F).
new1(A,B,C,D,E,F) :- G ≥ F+1, A = −2+C−G, B = 1+G, H = 1+G, I = 1+G, F−G ≤ 0,
    F ≥ 0, C−G ≥ 2, J−K ≥ 1, K−L ≥ 0, read(D,G,L), read(D,F,J), read(D,H,K),
    val(j,G), val(k,F), val(j,B), new2(A,G,C,D,E,F).
new2(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, B = 1+G, I = 1+G, H = 1+G, J = 1+G,
    K = 1+G, A−C+G ≤ −2, F−G ≤ 0, L−E ≤ −1, A ≥ 0, F ≥ 0, A−C+G ≥ −3, M−E ≥ 1,
    read(N,F,M), read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
    val(j,G), val(k,F), val(j,B), new4(A,G,C,N,P,F).
new2(A,B,C,D,E,F) :- G ≥ F+1, B = 1+G, H = 1+G, I = 1+G, A−C+G ≤ −2, F−G ≤ 0,
    A ≥ 0, F ≥ 0, A−C+G ≥ −3, J−K ≥ 1, K−L ≥ 0, read(D,G,L), read(D,F,J),
    read(D,H,K), val(j,G), val(k,F), val(j,B), new4(A,G,C,D,E,F).
new4(A,B,C,D,E,F) :- G ≥ F+1, H ≥ F+1, B = 1+G, I = 1+G, H = 1+G, J = 1+G,
    K = 1+G, A−C+G ≤ −2, F−G ≤ 0, L−E ≤ −1, A ≥ 0, F ≥ 0, M−E ≥ 1, read(N,F,M),
    read(N,K,L), read(N,G,E), write(O,H,E,D), write(N,G,L,O),
    val(j, G), val(k, F), val(j, B), new4(A, G, C, N, P, F).
new4(A, B, C, D, E, F) :- G ≥ F+1, B = 1+G, H = 1+G, I = 1+G, A−C+G ≤ −2, F−G ≤ 0,
    A ≥ 0, F ≥ 0, J−K ≥ 1, K−L ≥ 0, read(D, G, L), read(D, F, J), read(D, H, K),
    val(j, G), val(k, F), val(j, B), new4(A, G, C, D, E, F).
```

Since this set contains no constrained facts, by Removal of Useless Clauses we remove all clauses from $T2$ and the *Transform* strategy outputs the empty program. Thus, `incorrect` $\notin M(T2)$ and we conclude that the program *bubblesort-inner* is correct with respect to the given φ_{init} and φ_{error} properties.

4 Experimental Evaluation

We have implemented our verification method as a module of the VeriMAP software model checker [14] (available at `http://map.uniroma2.it/VeriMAP`) and we have performed an experimental evaluation of our method on a benchmark set of programs taken from the literature [6, 9, 16, 27, 35] (the source code is available at `http://map.uniroma2.it/smc/array-chr`).

We have applied the *Transform* strategy presented in Section 3 using different generalization strategies that combine the widening and convex hull operators together with various embedding relations. Different embedding relations are obtained: (i) by selecting different sets of variable identifiers for the introduction of the `val` constraints, and (ii) by using different relations to compare sets of identifiers (see Section 3.3). In particular, we have considered the following generalization strategies: $Gen_{W,\mathcal{I},\Cap}$, $Gen_{H,\mathcal{V},\subseteq}$, $Gen_{H,\mathcal{V},\Cap}$, $Gen_{H,\mathcal{I},\subseteq}$, and $Gen_{H,\mathcal{I},\Cap}$, where the subscripts should be interpreted as follows. The first subscript denotes the generalization operator: W stands for the widening operator, and H stands for the widening-and-convex-hull operator. The second subscript denotes the selected set of identifiers: $\mathcal{I}$ stands for the set of variable identifiers associated with the second argument (that is, the *index*) of the `read` constraints, and $\mathcal{V}$ stands for the set of identifiers associated with the third argument (that is, the *value*) of the `read` constraints. The third subscript denotes the relation $rel \in \{\subseteq, \Cap\}$ that is used for comparing the sets of identifiers.

The results of our experiments are summarized in Table 1. The experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under GNU/Linux OS. We have that the strategies based on $Gen_{H,\mathcal{I},rel}$ are more precise than those based on $Gen_{H,\mathcal{V},rel}$, for any $rel \in \{\subseteq, \Cap\}$. Similarly, the strategies based on $Gen_{H,S,\Cap}$ are more precise than those based on $Gen_{H,S,\subseteq}$, for any $S \in \{\mathcal{I}, \mathcal{V}\}$. Note that by generalizing the constraints, the *Transform* strategy may get an empty set of identifiers associated with a given variable, thereby making the generalizations based on the operator $\subseteq$ less useful that those based on the operator $\Cap$. The best trade-off between precision and performance is obtained by $Gen_{H,\mathcal{I},\Cap}$ that allowed us to prove all programs we have considered. Note also that the *bubblesort-inner* program can be proved only by generalizations based on $Gen_{W,\mathcal{I},\Cap}$ or $Gen_{H,\mathcal{I},\Cap}$.

5 Related Work and Conclusions

The technique presented in this paper is an extension of the one presented in [13]. The novel contributions of this paper are the following. (1) We have formalized

Program	$Gen_{W,\mathcal{I},\Cap}$	$Gen_{H,\mathcal{V},\subseteq}$	$Gen_{H,\mathcal{V},\Cap}$	$Gen_{H,\mathcal{I},\subseteq}$	$Gen_{H,\mathcal{I},\Cap}$
bubblesort-inner	0.9	*unknown*	*unknown*	*unknown*	1.52
copy-partial	*unknown*	*unknown*	3.52	3.51	3.54
copy-reverse	*unknown*	*unknown*	5.25	*unknown*	5.23
copy	*unknown*	*unknown*	5.00	4.88	4.90
find-first-non-null	0.14	0.66	0.64	0.28	0.27
find	1.04	6.53	2.35	2.33	2.29
first-not-null	0.11	0.22	0.22	0.22	0.22
init-backward	*unknown*	1.04	1.04	1.03	1.04
init-non-constant	*unknown*	2.51	2.51	2.47	2.47
init-partial	*unknown*	0.9	0.89	0.9	0.89
init-sequence	*unknown*	4.38	4.33	4.41	4.29
init	*unknown*	1.00	0.97	0.98	0.98
insertionsort-inner	0.58	2.41	2.4	2.38	2.37
max	*unknown*	*unknown*	0.8	0.81	0.82
partition	0.84	1.77	1.78	1.76	1.76
rearrange-in-situ	*unknown*	*unknown*	3.06	3.01	3.03
selectionsort-inner	*unknown*	*time-out*	*unknown*	2.84	2.83
precision	6	10	15	15	17
total time	3.61	21.42	34.76	31.81	38.45
average time	0.60	2.14	2.31	2.12	2.26

Table 1. Verification results using VeriMAP. Time is in seconds. By '*unknown*' we indicate that VeriMAP terminates without being able to prove correctness or incorrectness. By '*time-out*' we indicate that VeriMAP is unable to provide an answer within 5 minutes.

constraint replacement as a CHR program representing the Theory of Arrays, whereas in [13] constraint replacement was implemented directly in CLP. We have shown that the approach based on CHR allows a very elegant combination of constraint manipulation with transformations based on unfold/fold rules. (2) We have presented a novel strategy that controls the generalization of array constraints during CLP transformation by taking into account the information relating the variable identifiers in the imperative program and the CLP representation of their values. We have shown that our generalization strategy is effective on several examples taken from the literature.

In the Introduction we mentioned some CLP-based program verification methods. Here we briefly recall other methods, not based on CLP, for the verification of array programs.

Some of these methods use *abstract interpretation*. In [27], which builds upon [24], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is followed in [9] where a scalable framework for the automatic analysis of array programs is introduced. In [21, 34] a predicate abstraction for inferring universally quantified properties of array elements is presented, and in [26] the authors present a similar technique which uses template-based quantified abstract domains. In [46] a backward reachability analysis based on predicate abstraction and abstraction refinement is used for verifying assertions which are universally quantified over array indexes.

The methods based on abstract interpretation construct over-approximations, that is, invariants implied by the program executions. These methods have the advantage of being quite efficient because they fix in advance a finite set of basic assertions from which the invariants can be constructed. However, for the same reason, these methods may lack flexibility as the abstraction should be re-designed when verification fails.

Also *theorem provers* have been applied for discovering invariants and proving verification conditions generated from the programs. In particular, in [7] a satisfiability decision procedure for a decidable fragment of a theory of arrays is presented. That fragment is expressive enough to prove properties such as sortedness of arrays. In [32, 33, 38] the authors present some techniques that use theorem proving for generating array invariants. Some theorem proving techniques for program verification are based on *Satisfiability Modulo Theory* (SMT) (see, for instance, [3, 4, 35]). The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation because no finite set of assertions is fixed in advance and, instead, the suitable assertions needed for the proofs can be generated on demand.

Although the approach based on CLP program transformation shares many ideas and techniques with abstract interpretation and automated theorem proving, we believe that it offers a higher degree of flexibility and parametricity. Indeed, the transformation-based method for the generation of the verification conditions and their proof, is to a large extent independent of the imperative program and the property to be verified.

The use of CHR further enhances the flexibility of our transformation-based approach because CHR manipulate the constraints that represent operations on the data structures (such as the read and write operations in the case of arrays), while the unfold/fold rules manipulate the non-constraint atoms of the CLP programs. The experimental results we have reported in this paper demonstrate that the combination of the two kind of rules, those for constraints and those for non-constraint atoms, is a promising, powerful technique for proving program properties.

As future work we plan to extend our transformation-based method to the verification of programs which manipulate *dynamic data structures* such as lists or heaps. To this aim we may combine the CHR axiomatization of heaps proposed by [17] with the generalization strategies based on widening and convex-hull considered in this paper.

References

1. S. Abdennadher and H. Schütz. CHR$^\vee$: A flexible query language. *Proc. FQAS '98*, LNCS 1495, pages 1–14. Springer, 1998.
2. E. Albert, M. Gómez-Zamalloa, L. Hubert, and G. Puebla. Verification of Java bytecode using analysis and transformation of logic programs. *Proc. PADL '07*, LNCS 4354, pages 124–139. Springer, 2007.
3. F. Alberti, S. Ghilardi, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. *Proc. CAV '12*, LNCS 7358, pages 679–685. Springer, 2012.

4. F. Alberti, S. Ghilardi, and N. Sharygina. Decision procedures for flat array properties. *Proc. TACAS '14*, LNCS 8413, pages 15–30. Springer, 2014.
5. N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. *Proc. SMT-COMP '12*, pages 3–11, 2012.
6. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. *Proc. SAS '13*, LNCS 7935, pages 105–125. Springer, 2013.
7. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? *Proc. VMCAI '06*, volume *LNCS 3855*, pages 427–442. Springer, 2006.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. *Proc. POPL '77*, pages 238–252. ACM, 1977.
9. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *Proc. POPL '11*, pages 105–118. ACM, 2011.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *Proc. POPL '78*, pages 84–96. ACM, 1978.
11. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of imperative programs by constraint logic program transformation. *Proc. SAIRP '13*, EPTCS 129, pages 186–210, 2013.
12. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying programs via iterated specialization. *Proc. PEPM '13*, pages 43–52. ACM, 2013.
13. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying array programs by transforming verification conditions. *Proc. VMCAI '14*, LNCS 8318, pages 182–202. Springer, 2014.
14. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A tool for verifying programs through transformations. *Proc. TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014.
15. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 2014 (to appear).
16. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. *Proc. ESOP '10*, LNCS 6012, pages 246–266. Springer, 2010.
17. G. J. Duck, J. Jaffar, and N. C. H. Koh. Constraint-based program reasoning with heaps and separation. *Proc. CP '13*, LNCS 8124, pages 282–298. Springer, 2013.
18. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
19. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming*, 13(2):175–199, 2013.
20. C. Flanagan. Automatic software model checking via constraint logic. *Science of Computer Programming*, 50(1–3):253–270, 2004.
21. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. *Proc. POPL '02*, pages 191–202. ACM, 2002. ACM.
22. T. Frühwirth. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, pages 95–138, October 1998.
23. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.
24. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. *Proc. POPL '05*, pages 338–350. ACM, 2005.

25. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A software verifier based on Horn clauses. *Proc. TACAS '12*, LNCS 7214, pages 549–551. Springer, 2012.
26. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. *Proc. TACAS '08*, LNCS 4963, pages 443–458. Springer, 2008.
27. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. *Proc. PLDI '08*, pages 339–348. ACM, 2008.
28. K. S. Henriksen and J. P. Gallagher. Abstract interpretation of PIC programs through logic programming. *Proc. SCAM '06*, pages 103–179, 2006.
29. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.
30. J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. http://paella.d1.comp.nus.edu.sg/tracer/, 2012.
31. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. *Proc. CP '09*, LNCS 5732, pages 454–469. Springer, 2009.
32. R. Jhala and K. L. McMillan. Array abstractions from proofs. *Proc. CAV '07*, LNCS 4590, pages 193–206. Springer, 2007.
33. L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. *Proc. FASE '09*, LNCS 5503, pages 470–485. Springer, 2009.
34. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.
35. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. *Proc. VMCAI '13*, LNCS 7737, pages 169–188. Springer, 2013.
36. J. W. Lloyd. *Foundations of logic programming*. Springer-Verlag, Berlin, 1987. Second edition.
37. J. McCarthy. Towards a mathematical science of computation. *Proc. IFIP 1962*, pages 21–28. North Holland, 1963.
38. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. *Proc. TACAS '08*, LNCS 4963, pages 413–427. Springer, 2008.
39. M. Méndez-Lojo, J. A. Navas, and M. V. Hermenegildo. A flexible, (C)LP-based approach to the analysis of object-oriented programs. *Proc. LOPSTR '07*, LNCS 4915, pages 154–168. Springer, 2008.
40. J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. *Proc. LOPSTR '02*, LNCS 2664, pages 90–108. Springer, 2003.
41. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of imperative programs through analysis of constraint logic programs. *Proc. SAS '98*, LNCS 1503, pages 246–261. Springer, 1998.
42. A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
43. A. Podelski and A. Rybalchenko. ARMC: The logical choice for software model checking with abstraction refinement. *Proc. PADL '07*, LNCS 4354, pages 245–259. Springer, 2007.
44. C. J. Reynolds. *Theories of programming languages*. Cambridge University Press, 1998.
45. P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. *Proc. CAV '13*, LNCS 8044, pages 347–363. Springer, 2013.
46. M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. *Proc. SAS '09*, LNCS 5673, pages 3–18. Springer, 2009.