Fundamenta Informaticae Submitted (2015) 1–26
IOS Press

A Rule-based Verification Strategy for Array Manipulating Programs

Emanuele De Angelis

University of Chieti-Pescara, Pescara, Italy, emanuele.deangelis@unich.it

Fabio Fioravanti

University of Chieti-Pescara, Pescara, Italy, fioravanti@unich.it

Alberto Pettorossi

University of Rome Tor Vergata, Via del Politecnico 1, 00133 Rome, Italy, pettorossi@disp.uniroma2.it

Maurizio Proietti

IASI-CNR, Viale Manzoni 30, 00185 Rome, Italy, maurizio.proietti@iasi.cnr.it

Abstract. We present a method for verifying properties of imperative programs that manipulate integer arrays. Imperative programs and their properties are represented by using Constraint Logic Programs (CLP) over integer arrays. Our method is refutational. Given a Hoare triple $\{\varphi\}$ *prog* $\{\psi\}$ that defines a partial correctness property of an imperative program *prog*, we encode the negation of the property as a predicate `incorrect` defined by a CLP program P, and we show that the property holds by proving that `incorrect` is *not* a consequence of P. Program verification is performed by applying a sequence of semantics preserving transformation rules and deriving a new CLP program T such that `incorrect` is a consequence of P iff it is a consequence of T. The rules are applied according to an automatic strategy whose objective is to derive a program T that satisfies one of the following properties: either (i) T is the empty set of clauses, hence proving that `incorrect` does not hold and *prog* is correct, or (ii) T contains the fact `incorrect`, hence proving that *prog* is incorrect. Our transformation strategy makes use of an axiomatization of the theory of arrays for the manipulation of array constraints, and also applies the widening and convex hull operators for the generalization of linear integer constraints. The strategy has been implemented in the VeriMAP transformation system and it has been shown to be quite effective and efficient on a set of benchmark array programs taken from the literature.

1. Introduction

Many methods have been proposed in the literature for verifying and proving properties of C-like, imperative programs using the formalism of Constraint Logic Programming (CLP).

Some of those methods follow the approach initially presented in [44], which is based on program specialization and abstract interpretation [7]. The first step of that approach consists in encoding as a CLP program the interpreter of the imperative language in which programs are written, and then, in the second step, this CLP program is specialized with respect to the imperative program under investigation, thereby deriving a new CLP program. Finally, in the third step, this new CLP program is analyzed by computing an overapproximation of its least model by a bottom-up evaluation of an abstraction of the program [1, 27, 41], and that analysis is used to prove (or disprove) the property of interest.

Other program verification methods start off from a partial correctness triple of the form $\{\varphi\}$*prog*$\{\psi\}$ and from that triple they generate a CLP program, called the *verification conditions* for *prog* [5, 50] and here denoted by *VC*, by using ad hoc algorithms which take into account the semantics of the imperative language in which *prog* is written. The CLP program *VC* does not contain any explicit reference to the imperative program *prog*. Then, from *VC* one can infer the validity of the given triple by using goal directed, symbolic evaluation together with other techniques such as *interpolation* [14, 18, 30, 31].

In order to infer the validity of a given triple, various other reasoning techniques can be applied to the CLP program *VC*. In particular, the techniques presented in [4, 23, 46, 48] (where CLP programs are also called *constrained Horn clauses*), make use of *CounterExample-Guided Abstraction Refinement* (CEGAR) and *Satisfiability Modulo Theories* (SMT).

In this paper we follow the verification approach for imperative programs based on transformations of CLP programs which has been presented in [10, 12]. Given a partial correctness property expressed by the triple $\{\varphi\}$ *prog* $\{\psi\}$, we first encode the negation of that property as a predicate `incorrect` defined by a CLP program P. Then, similarly to [44], we specialize the CLP program P with respect to the CLP representation of the imperative program *prog* and we generate a new CLP program *VC* representing the verification conditions for *prog*. At this point our verification method departs from the one presented in [44] and all other verification methods mentioned above. Indeed, the final step of our verification method consists in the application to *VC* of a sequence of equivalence preserving transformations with the objective of deriving a CLP program T such that either (i) T is the empty set of clauses, hence proving that `incorrect` does not hold and *prog* is correct, or (ii) T contains the fact `incorrect`, hence proving that *prog* is incorrect.

Due to the undecidability of the partial correctness problem, it may be the case that from the program *VC*, using our verification method, we derive a CLP program containing one or more clauses of the form `incorrect :- G`, where `G` is a non-empty conjunction, and we can conclude neither that *prog* is correct nor that *prog* is incorrect. However, despite the possibility for these inconclusive answers, our verification method performs well in practice, as experimentally shown in Section 6.

The main contributions of this paper are the following.
(1) We provide a method that given any partial correctness triple $\{\varphi\}$*prog*$\{\psi\}$, where *prog* is an imperative program that manipulates integers and integer arrays, generates the verification conditions for *prog*. In those verification conditions the read and write operations on arrays are represented as constraints.
(2) We show how the verification conditions can be manipulated by using the familiar *unfold/fold transformation rules* for CLP programs [15]. The transformation rules include a *constraint replacement* rule which is used for manipulating the read and write constraints on arrays.
(3) We propose a *transformation strategy* for guiding the application of the transformation rules, with the objective of transforming verification conditions and proving the validity of the given triple. In particular, we design a novel *generalization strategy for array constraints* for the introduction, during the transformation of the CLP programs, of the new predicate definitions required for the verification of the properties of interest. These new predicate definitions correspond to the invariants holding throughout the execution of the imperative programs. Our generalization strategy makes use of operators, such as the widening and convex hull operators, that have been introduced in the field of *abstract interpretation* [9] and extends to CLP(Array) programs the generalization strategies considered in [16, 17] for CLP

programs over the integers.
(4) Finally, through an experimental evaluation based on a prototype implementation that uses the VeriMAP transformation system [11], we demonstrate that our verification method performs well on a set of benchmark programs taken from the literature.

The paper is structured as follows. In Section 2 we introduce the class of CLP(Array) programs, that is, logic programs with constraints over the integers and integer arrays. In Section 3 we present the unfolding/folding rules including the constraint replacement rule for manipulating constraints over integer arrays [6, 20, 39]. Then, in Section 4 we show how to generate the verification conditions via the specialization of CLP(Array) programs. In Section 5 we present an automatic strategy for guiding the application of the transformation rules with the objective of proving (or disproving) a given property of interest. Finally, in Section 6 we present various experimental results obtained by using our VeriMAP verification system [11].

This paper is an improved, extended version of [10]. Here we present formal soundness, termination, and confluence results, and a more extensive experimental comparison with related techniques.

2. CLP(Array): Constraint Logic Programs on Arrays

In this section we recall some basic notions concerning Constraint Logic Programming (CLP), and we introduce the set, called CLP(Array), of CLP programs with constraints over the integers and the integer arrays. Other notions concerning CLP can be found in [29]. For reasons of simplicity, in this paper we will consider one-dimensional arrays only. We leave it for future investigation the case of multi-dimensional arrays.

Atomic integer constraints are formulas of the form: $\mathtt{p}_1\!=\!\mathtt{p}_2$, 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. When writing polynomials, the symbols `+` and `*` denote sum and multiplication, respectively. An *integer constraint* is a conjunctions of atomic integer constraints. *Atomic array constraints* are constraints of the form: `dim(A,N)`, denoting that the array `A` has dimension `N`, or `read(A,I,V)`, denoting that the `I`-th element of the array `A` is the value `V`, or `write(A,I,V,B)`, denoting that the array `B` is equal to the array `A`, except that its `I`-th element is the value `V`. We assume that indexes of arrays and elements of arrays are integers. An *array constraint* is a conjunctions of atomic array constraints. A *constraint* is either `true`, or `false`, or an integer constraint, or an array constraint, or a conjunction of constraints.

An *atom* is an atomic formula of the form: $\mathtt{q(t_1,\ldots,t_m)}$, where `q` is a predicate symbol not in $\{=, \geq, >, \mathtt{dim}, \mathtt{read}, \mathtt{write}\}$, and $\mathtt{t}_1, \ldots, \mathtt{t_m}$ are terms constructed out of variables, constants, and function symbols different from `+` and `*`. A CLP(Array) 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` and the conjunction `c, B` are called the *head* and the *body* of the clause, respectively. Without loss of generality, we assume that in every clause head, all occurrences of integer terms are distinct variables. For instance, the clause `p(X,X+1) :- X>0, q(X)` will be written as `p(X,Y) :- Y=X+1, X>0, q(X)`. 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(Array) 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.

We say that a predicate `p` *depends on* a predicate `q` in a program P if either in P there is a clause of the form `p(...) :- c, B` such that `q` occurs in `B`, or there exists a predicate `r` such that `p` depends on `r` in P and `r` depends on `q` in P. By *vars*(φ) we denote the set of all free variables of the formula φ.

Now we define the semantics of CLP(Array) programs. An $\mathcal{A}$-*interpretation* I is an interpretation such that:

(i) the carrier of I is the Herbrand universe [38] constructed out of the set $\mathbb{Z}$ of the integers, the finite sequences of integers (which provide the interpretation for arrays), 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 $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ of integers, for all integers 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 i and 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 [38] 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. If a formula φ is true in every $\mathcal{A}$-interpretation we write $\mathcal{A} \models \varphi$, and we say that φ is true in $\mathcal{A}$. A constraint c is *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where for every formula φ, $\exists(\varphi)$ denotes the existential closure of φ. Likewise, $\forall(\varphi)$ denotes the universal closure of φ. A constraint is *unsatisfiable* if it is not satisfiable. A constraint c *entails* a constraint d, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. Given any two integer constraints $\mathtt{i}_1$ and $\mathtt{i}_2$, we will feel free to write $\mathbb{Z} \models \forall(\mathtt{i}_1 \leftrightarrow \mathtt{i}_2)$, instead of $\mathcal{A} \models \forall(\mathtt{i}_1 \leftrightarrow \mathtt{i}_2)$. Given a constraint c, we write $\mathtt{c}{\downarrow}\mathbb{Z}$ to denote the conjunction of all the integer constraints occurring in c.

The semantics of a CLP(Array) program P is defined to be the *least* $\mathcal{A}$-*model* of P, denoted $M(P)$, that is, the least $\mathcal{A}$-interpretation I such that every clause of P is true in I.

3. Transformation Rules for CLP(Array) Programs

Our verification method is based on the application of some transformation rules that preserve the least $\mathcal{A}$-model semantics of CLP(Array) programs. In particular, we apply the following *transformation rules*, collectively called *unfold/fold rules*: (i) *Definition*, (ii) *Unfolding*, (iii) *Constraint Replacement*, and (iv) *Folding*. These rules are an adaptation to CLP(Array) programs of the unfold/fold rules for a generic CLP language (see, for instance, [15]). The soundness of the rules we consider is proved in [15].

Let P be any given CLP(Array) program.

(i) *Definition Rule.* By the definition rule we introduce a clause of the form `newp(X) :- c,A`, where `newp` is a new predicate symbol (occurring neither in P nor in a clause previously introduced by the definition rule), X is the tuple of variables occurring in the atom A, and c is a constraint.

(ii) *Unfolding Rule.* Let us consider a clause C of the form `H :- c,L,A,R`, where H and A are atoms, c is a constraint, and L and R are (possibly empty) conjunctions of atoms. Let us also consider the set $\{\mathtt{K}_i \text{ :- } \mathtt{c}_i, \mathtt{B}_i \mid i=1,\ldots,m\}$ of the (renamed apart) clauses of P such that, for $i=1,\ldots,m$, A is unifiable with $\mathtt{K}_i$ via the most general unifier ϑ_i and $(\mathtt{c}, \mathtt{c}_i)\,\vartheta_i$ is satisfiable. By unfolding C w.r.t. A using P, we derive the set $\{(\mathtt{H} \text{ :- } \mathtt{c}, \mathtt{c}_i, \mathtt{L}, \mathtt{B}_i, \mathtt{R})\,\vartheta_i \mid i=1,\ldots,m\}$ of clauses.

(iii) *Constraint Replacement Rule.* Let us consider a clause C of the form: `H :- c`$_0$, B, and some constraints $\mathtt{c}_1, \ldots, \mathtt{c}_\mathtt{n}$ such that

$\mathcal{A} \models \forall\,((\exists \mathtt{X}_0\, \mathtt{c}_0) \leftrightarrow (\exists \mathtt{X}_1\, \mathtt{c}_1 \vee \ldots \vee \exists \mathtt{X}_\mathtt{n}\, \mathtt{c}_\mathtt{n}))$

where, for $i=0,\ldots,\mathtt{n}$, $\mathtt{X_i} = \mathit{vars}(\mathtt{c_i}) - \mathit{vars}(\mathtt{H},\mathtt{B})$. Then, by constraint replacement from clause C we derive $\mathtt{n}$ clauses $C_1,\ldots,C_\mathtt{n}$ obtained by replacing in the body of C the constraint $\mathtt{c_0}$ by the $\mathtt{n}$ constraints $\mathtt{c_1},\ldots,\mathtt{c_n}$, respectively.

The equivalences, also called the *Laws of Arrays*, needed for applying the constraint replacement rule can be shown to be true in $\mathcal{A}$ by using (a relational version of) the theory of arrays with dimension [6, 20]. This theory includes the following axioms, where all variables are universally quantified at the front:

(A1) $\mathtt{I}=\mathtt{J},\ \mathtt{read(A,I,U)},\quad \mathtt{read(A,J,V)} \ \rightarrow\ \mathtt{U}=\mathtt{V}$

(A2) $\mathtt{I}=\mathtt{J},\ \mathtt{write(A,I,U,B)},\ \mathtt{read(B,J,V)} \ \rightarrow\ \mathtt{U}=\mathtt{V}$

(A3) $\mathtt{I}\neq\mathtt{J},\ \mathtt{write(A,I,U,B)},\ \mathtt{read(B,J,V)} \ \rightarrow\ \mathtt{read(A,J,V)}$

Axiom (A1) is often called *array congruence*, and Axioms (A2) and (A3) are collectively called *read-over-write*. We do not list here the obvious axioms that state that the array indexes of the read and write operations are within the bounds specified by the predicate `dim`.

(iv) *Folding Rule.* Given a clause E: `H :- e, L, A, R` and a clause D: `K :- d, D` introduced by the definition rule. Suppose that, for some substitution ϑ, (i) $\mathtt{A} = \mathtt{D}\,\vartheta$, and (ii) $\forall\,(\mathtt{e} \rightarrow \mathtt{d}\,\vartheta)$. Then by folding E using D we derive $\mathtt{H\ {:}\text{-}\ e,\ L,\ K}\,\vartheta\mathtt{,\ R}$.

From P we can derive a new program *TransfP* by: (i) selecting a clause C in P, (ii) deriving a new set *TransfC* of clauses by applying one or more transformation rules, and (iii) replacing C by *TransfC* in P. Clearly, we can apply a new sequence of transformation rules starting from *TransfP* and iterate this process at will.

The following theorem is an immediate consequence of the soundness results for the unfold/fold transformation rules of CLP programs [15].

Theorem 3.1. (*Soundness of the Transformation Rules*) Let the CLP(Array) program *TransfP* be derived from P by a sequence of applications of the transformation rules. Suppose that every clause introduced by the definition rule is unfolded at least once in this sequence. Then, for every ground atom A in the language of P, $A \in M(P)$ iff $A \in M(\mathit{TransfP})$.

The assumption that the unfolding rule should be applied at least once, is required for technical reasons [15]. Informally, that assumption forbids the replacement of a definition clause of the form `A:-B` by the clause `A:-A` that is obtained by folding clause `A:-B` using `A:-B` itself. Indeed, a similar replacement in general does not preserve the least $\mathcal{A}$-model semantics.

4. Generating Verification Conditions via Specialization

We consider a C-like imperative programming language with integer and array variables, assignments (`=`), sequential compositions (`;`), conditionals (`if-else`), while-loops (`while`), and jumps (`goto`). A program is a sequence of (labeled) commands. We assume that in each program there is a unique initial command with label ℓ_0 and a unique `halt` command with label ℓ_h which, when executed, causes the program to terminate.

The semantics of the imperative language considered here is defined by means of a *transition relation*, denoted $\Longrightarrow$, between *configurations*. Each configuration is a pair $\langle\!\langle c,\delta\rangle\!\rangle$ of a command c and an *environment* δ. An environment δ is a function 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 definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [47] and is omitted. We say that a configuration $\langle\!\langle c, \delta \rangle\!\rangle$ satisfies a property φ whose free variables are $z_1, \ldots, z_r$ iff $\varphi(\delta(z_1), ..., \delta(z_r))$ is true in $\mathcal{A}$.

We find it convenient to define the partial correctness of a program by considering the *negation* of the postcondition of the program. We say that the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$ is *valid*, meaning that *prog* is *partially correct* (or, simply, *correct*) with respect to the given precondition and postcondition, if for all terminating executions of *prog* starting from an input satisfying φ_{init}, the output satisfies $\neg\varphi_{error}$. In other words, *prog* is *incorrect* if there exists an execution of *prog* that leads from a configuration satisfying the property φ_{init} and whose command is the initial command (also called an *initial configuration*), to a configuration whose command is `halt` and whose environment satisfies the property φ_{error} (also called an *error configuration*). In this paper we assume that φ_{init} and φ_{error} are formulas of the form: $\exists x_1 \ldots \exists x_n.c$, where c is a constraint and the free variables of $\exists x_1 \ldots \exists x_n.c$ are global variables occurring in *prog*.

Obviously, when writing a Hoare triple we may use a less restrictive syntax, as long as the triple can be translated into one or more triples of the form specified above. For example, in Section 6 we wrote the triple for the *copy* program as: $\{true\}$ *copy* $\{\forall i.\ (0 \leq i \wedge i < n) \rightarrow a[i] = b[i]\}$ and this is a legal triples because it can be translated into the conjunction of the following two triples:
(1) $\{true\}$ *copy* $\{\neg\exists i.\ 0 \leq i \wedge i < n \wedge a[i] > b[i]\}$ (2) $\{true\}$ *copy* $\{\neg\exists i.\ 0 \leq i \wedge i < n \wedge a[i] < b[i]\}$
It follows directly from the definitions that the problem of checking whether or not *prog* is incorrect can be encoded as the problem of checking whether or not the nullary predicate `incorrect` is a consequence of the CLP(Array) program P made out of the following clauses:

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

together with the clauses for the predicates (i) `initConf(X)`, (ii) `errorConf(X)`, and (iii) `tr(X,Y)`. These three predicates encode an initial configuration, an error configuration, and the transition relation $\Longrightarrow$ between configurations, respectively. The predicate `reach(Y)` holds if a configuration `Y` can be reached from an initial configuration. Note that the existential quantifiers possibly occurring in the two formulas φ_{init} and φ_{error} are dropped when these formulas are used in the body of the definition of `initConf(X)` and `errorConf(X)`, respectively, thereby obtaining CLP(Array) clauses.

As an example of the clauses defining the predicate `tr`, let us present the following clause encoding the transition relation for the labeled command $\ell\colon a[ie] = e$ that assigns the value of e to the element of index ie of the array a (here a configuration of the form $\langle\!\langle \ell : c, \delta \rangle\!\rangle$, where c is a command with label ℓ and δ is an environment, is denoted by the term `cf(cmd(L,C), D)`):

```
tr(cf(cmd(L,asgn(arrayelem(A,IE),E)),D), cf(cmd(L1,C),D1)) :-
    eval(IE,D,I), eval(E,D,V), lookup(D,A,FA), write(FA,I,V,FA1),
    update(D,A,FA1,D1), nextlab(L,L1), at(L1,C).
```

In this clause: (i) `arrayelem(A,IE)` is a term representing the expression $a[ie]$, (ii) `eval(IE,D,I)` holds iff `I` is the the value of the index expression `IE` in the environment `D`, (iii) `eval(E,D,V)` holds iff `V` is the value of the expression `E` in the environment `D`, (iv) `lookup(D,A,FA)` holds iff `FA` is the value of the array `A` in the environment `D`, (v) `update(D,A,FA1,D1)` holds iff `D1` is the environment `D` after the assignment to array `A` producing the new array `FA1`, (vi) `nextlab(L,L1)` holds iff `L1` is the label of the command following the command with label `L` in the encoding of the given program *prog*, and (vii) `at(L1,C)` holds iff `C` is the command with label `L1` in that encoding. As shown in [12], similar clauses for the predicate `tr` can be defined for the other commands of the imperative language we consider.

The Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$ is valid iff `incorrect` $\notin M(P)$, where $M(P)$ is the least $\mathcal{A}$-model of program P (see Section 2).

Our verification method consists of the following two steps, each of which is performed by a sequence of applications of the unfold/fold transformation rules presented in Section 3, starting from program P:
(i) the *Generation of Verification Conditions* (*VCGen*), outlined below, and
(ii) the *Transformation of Verification Conditions* (*VCTransf*), which will be presented in the next section.
Since the rules preserve the least $\mathcal{A}$-model (see Theorem 3.1), we will have that `incorrect` $\in M(P)$ iff `incorrect` $\in M(T)$, where T is the CLP program derived after applying *VCGen* and *VCTransf*.

During *VCGen*, program P is specialized with respect to the predicate definitions of `tr` (which depends on *prog*), `initConf`, and `errorConf`, thereby deriving a new program, called the *verification conditions* for *prog* and denoted by VC, which does not contain any occurrence of those predicates and has no reference to the commands of the imperative program *prog*. For this reason, program VC is said to be derived by applying *the removal of the interpreter* (see, for instance, [12]).

We say that program VC is *satisfiable* iff `incorrect` $\notin M(VC)$. Thus, the satisfiability of the verification conditions for *prog* guarantees that the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$ is valid.

In our verification method, the specialization of P is done by applying a variant of the removal of the interpreter strategy presented in [12]. The main difference with respect to [12] is that the CLP programs we consider here may contain `read`, `write`, and `dim` predicates. The `read` and `write` predicates are never unfolded during the specialization and they occur in the residual CLP(Array) program VC. Moreover all occurrences of the `dim` predicate are eliminated by replacing them by suitable integer constraints on indexes. We do not show here the *VCGen* step, and we refer to [12] for a detailed presentation in the case of programs without array operations. Here we only show an example of generation of the verification conditions.

Let us consider the following program *SeqInit* which initializes a given array a of n integers by the sequence: $a[0],\ a[0]{+}1,\ \ldots,\ a[0]{+}n{-}1$:

SeqInit: ℓ_0: $i = 1;$
ℓ_1: `while` $(i{<}n)$ $\{\ a[i] = a[i{-}1] + 1;\ i = i + 1;\ \};$
ℓ_h: `halt`

We consider the Hoare triple $\{\varphi_{init}(i,n,a)\}$ *SeqInit* $\{\neg\varphi_{error}(n,a)\}$, where:
(i) $\varphi_{init}(i,n,a)$ is $i \geq 0\ \wedge\ n = dim(a)\ \wedge\ n \geq 1$, and
(ii) $\varphi_{error}(n,a)$ is $\exists j.\ j \geq 0\ \wedge\ j \leq n{-}2\ \wedge\ a[j] \geq a[j{+}1]$.

First, the above triple is translated into a CLP(Array) program P. In particular, the properties φ_{init} and φ_{error} are defined by the following clauses, respectively:

1. `phiInit(I,N,A) :- I≥0, dim(A,N), N≥1.`
2. `phiError(N,A) :- K=J+1, J≥0, J≤N−2, U≥V, read(A,J,U), read(A,K,V).`

The clauses defining the predicates `initConf` and `errorConf` which specify the initial and the error configurations, respectively, are as follows:

3. `initConf(cf(cmd(l0,Cmd),[(i,I),(n,N),(a,A)])) :- at(l0,Cmd), phiInit(I,N,A).`
4. `errorConf(cf(cmd(lh,Cmd),[(i,I),(n,N),(a,A)])) :- at(lh,Cmd), phiError(N,A).`

where the environment is a finite function encoded as a list of (identifier, value) pairs. (In particular, `I` and `N` range over integers, and `A` ranges over sequences of integers.)

In order to encode the program *SeqInit*, we first replace the while-loop command by a conditional and a jump command, and then we introduce the following facts defining the predicate `at`:

```
at(l0, asgn(i, 1)).                    at(l1, ite(less(i, n), l2, lh)).
at(l2, asgn(arrayelem(a, i), plus(arrayelem(a, minus(i, 1)), 1))).
at(l3, asgn(i, plus(i, 1))).           at(l4, goto(l1)).              at(lh, halt).
```

Now we perform the *VCGen* step and from program P we obtain the following program VC:

```
5. incorrect :- K=J+1, J≥0, J≤N−2, U≥V, N≤I, read(A, J, U), read(A, K, V), p(I, N, A).
6. p(I, N, A) :- 1≤H, H≤N−1, G=H−1, I=H+1, Z=W+1, read(B,G,W), write(B,H,Z,A), p(H,N,B).
7. p(I, N, A) :- I=1, N≥1.
```

which represents the verification conditions for *SeqInit*.

The following Theorem 4.1 is a straightforward extension to the case of CLP(Array) programs of the results shown in [12].

Theorem 4.1. (*Termination and Soundness of the VCGen Transformation*) Let P be the CLP(Array) program defining the predicate `incorrect` which holds iff the triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$ is not valid. The *VCGen* transformation terminates on the input P, and derives a CLP(Array) program VC such that `incorrect` $\in M(P)$ iff `incorrect` $\in M(VC)$. Moreover, VC is a linear CLP(Array) program.

5. A Strategy for Transforming the Verification Conditions

In order to check whether or not `incorrect` $\in M(VC)$, the standard evaluation methods are often inadequate, because the least $\mathcal{A}$-model $M(VC)$ may be infinite and both the bottom-up and the top-down evaluation of the predicate `incorrect` may not terminate (indeed, this is the case in our *SeqInit* program above).

In this section, we present the *VCTransf* transformation step, which propagates the constraints occurring in φ_{init} and φ_{error} with the objective of deriving from program VC a new program T where the predicate `incorrect` is defined by either (i) the fact `incorrect` (in which case the verification conditions are unsatisfiable, that is, `incorrect` $\in M(VC)$, and *prog* is incorrect), or (ii) the empty set of clauses (in which case the verification conditions are satisfiable, that is, `incorrect` $\notin M(VC)$, and *prog* is correct). In the case where neither (i) nor (ii) holds, we cannot conclude anything about the correctness of *prog*. However, similarly to what has been proposed in [10], we can iterate a few times the *VCTransf* step in the hope of deriving a program where either (i) or (ii) holds. Obviously, due to undecidability limitations, it may be the case that we never get a program where either (i) or (ii) holds.

VCTransf is performed by applying the unfold/fold transformation rules according to the strategy shown in Figure 1. *VCTransf* can be viewed as a backward propagation of the constraints in φ_{error}. The forward propagation of the constraints in φ_{init} can be obtained by combining *VCTransf* with the *Reversal* transformation described in [10].

Let us describe in more detail the UNFOLDING, CONSTRAINT REPLACEMENT, and DEFINITION & FOLDING phases of the *VCTransf* strategy.

```
Input: A linear CLP(Array) program $VC$.
Output: Program $T$ such that incorrect $\in M(VC)$ iff incorrect $\in M(T)$.
INITIALIZATION: Let InDefs be the set of all clauses of $VC$ whose head is the atom incorrect;
$T := \emptyset$;   $Defs := InDefs$;
while in InDefs there is a clause $C$ do
    UNFOLDING: Unfold $C$ w.r.t. the single atom in its body by using $VC$, and derive a set $U(C)$ of
        clauses;
    CONSTRAINT REPLACEMENT: Apply a sequence of constraint replacements based on the Laws of
        Arrays, and derive from $U(C)$ a set $R(C)$ of clauses;
    CLAUSE REMOVAL: Remove from $R(C)$ all clauses whose body has an unsatisfiable constraint;
    DEFINITION & FOLDING: Introduce a (possibly empty) set of new predicate definitions and add
        them to Defs and to InDefs;
        Fold the clauses in $R(C)$ different from constrained facts by using the clauses in Defs, and derive
        a set $F(C)$ of clauses;
    $InDefs := InDefs - \{C\}$;   $T := T \cup F(C)$;
end-while;
REMOVAL OF USELESS CLAUSES: Remove from program $T$ all clauses with head predicate p, if in $T$
there is no constrained fact q(...) :- c where q is either p or a predicate on which p depends.
```

Figure 1. The *VCTransf* transformation strategy.

5.1. Unfolding

The UNFOLDING phase corresponds to one inference step, in a backward way, starting from the error configuration. For instance, let us consider again the *SeqInit* program of Section 4, and let VC be the CLP program made out of clauses 5, 6, and 7. The *VCTransf* strategy starts off by unfolding clause 5 w.r.t. the atom `p(I,N,A)`. We get the clause:

8. `incorrect :- K=J+1, J≥0, J≤N−2, U≥V, N≤I, 1≤H, H≤N−1, G=H−1, I=H+1, Z=W+1,`
`read(A,J,U), read(A,K,V), read(B,G,W), write(B,H,Z,A), p(H,N,B).`

where `B` denotes the array from which the output array `A` computed by the *SeqInit* program is derived. Basically, the body of clause 8 represents the set of configurations from which the error configuration is reachable. Note that the unfolding rule derives one clause only, because the conjunction of the constraint `c1` occurring in clause 5 and the constraint `c2` occurring in clause 7 is unsatisfiable (that is, the initial configuration is not backward reachable in one step from the error configuration).

5.2. Constraint Replacement

The CONSTRAINT REPLACEMENT transformation phase applies the Laws of Arrays and infers new constraints on the variables of the single atom that occurs in the body of each clause derived at the end of the UNFOLDING phase. The objective of CONSTRAINT REPLACEMENT is to simplify the array constraints and, in particular, to replace `read` constraints in favor of integer constraints (see rules RR1 and WR1). CONSTRAINT REPLACEMENT also performs, whenever possible, case reasoning on the array indexes (see clauses (α) and (β) of rule WR3).

This transformation phase works as follows. We select a clause in the set $U(C)$ of the clauses obtained by unfolding, and we replace it by the clause(s) obtained by applying *as long as possible* the following rules RR1–WR3, which are based on axioms A1–A3 of Section 3.

Let $\mathtt{H :- k, G}$ be a clause where $\mathtt{k} \equiv (\mathtt{c}, \mathtt{read(A, I, U)}, \mathtt{read(A, J, V)})$, $\mathtt{c}$ is a constraint, and $\mathtt{G}$ is a conjunction of atoms.

(RR1) *If* $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}=\mathtt{J})$ *then* replace $\mathtt{k}$ by $(\mathtt{c},\ \mathtt{U}=\mathtt{V},\ \mathtt{read(A, I, U)})$.

(RR2) *If* $\mathtt{k}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\neq\mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U}\neq\mathtt{V})$ *then* add to $\mathtt{k}$ the constraint $\mathtt{I}\neq\mathtt{J}$.

Let $\mathtt{H :- k, G}$ be a clause where $\mathtt{k} \equiv (\mathtt{c}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)})$, $\mathtt{c}$ is a constraint, and $\mathtt{G}$ is a conjunction of atoms.

(WR1) *If* $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}=\mathtt{J})$ *then* replace $\mathtt{k}$ by $(\mathtt{c},\ \mathtt{U}=\mathtt{V},\ \mathtt{write(A, I, U, B)})$.

(WR2) *If* $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}\neq\mathtt{J})$ *then* replace $\mathtt{k}$ by $(\mathtt{c},\ \mathtt{write(A, I, U, B)},\ \mathtt{read(A, J, V)})$.

(WR3) *If* $\mathtt{k}{\downarrow}\mathbb{Z} \not\sqsubseteq \mathtt{I}=\mathtt{J}$ and $\mathtt{k}{\downarrow}\mathbb{Z} \not\sqsubseteq \mathtt{I}\neq\mathtt{J}$ *then* replace $\mathtt{H :- k, G}$ by the two clauses:

(α) $\mathtt{H :- c,\ I=J,\ U=V,\ write(A, I, U, B),\ G}$

and (β) $\mathtt{H :- c,\ I\neq J,\ write(A, I, U, B),\ read(A, J, V),\ G}$

The replacement process, which in general is nondeterministic, is confluent and terminating, as stated in the following theorem.

Theorem 5.1. (*Soundness, Termination, and Confluence of Constraint Replacement*)
(1. *Soundness*) Each rule among RR1, RR2, WR1, WR2, and WR3 is a sound application of the constraint replacement rule. Indeed,
(i) if $\mathtt{H :- c_0, G}$ is replaced by $\mathtt{H :- c_1, G}$, by using a rule among RR1, RR2, WR1, WR2, then
$\mathcal{A} \models \forall(\mathtt{c_0} \leftrightarrow \mathtt{c_1})$, and
(ii) if $\mathtt{H :- c_0, G}$ is replaced by $\mathtt{H :- c_1, G}$ and $\mathtt{H :- c_2, G}$, by using WR3, then $\mathcal{A} \models \forall(\mathtt{c_0} \leftrightarrow \mathtt{c_1} \vee \mathtt{c_2})$.
(2. *Termination*) Let D be a clause obtained after the UNFOLDING phase of the *VCTransf* strategy. Then, the execution of the CONSTRAINT REPLACEMENT phase on D terminates.
(3. *Confluence*) The rules RR1, RR2, WR1, WR2, and WR3 are confluent, modulo equivalence of integer constraints.

Proof:
(1. *Soundness*) (i). The proof proceeds by cases on the rule used. Suppose that $\mathtt{H :- k, G}$ is replaced by $\mathtt{H :- k_1, G}$ by using RR1, where $\mathtt{k}$ is of the form $(\mathtt{c}, \mathtt{read(A, I, U)}, \mathtt{read(A, J, V)})$, $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}=\mathtt{J})$ holds, and $\mathtt{k_1}$ is of the form $(\mathtt{c}, \mathtt{U} = \mathtt{V}, \mathtt{read(A, J, V)})$. From axiom A1 of Section 3, we derive:

$$\mathcal{A} \models \forall(\mathtt{c}, \mathtt{read(A, I, U)}, \mathtt{read(A, J, V)} \leftrightarrow \mathtt{c}, \mathtt{U} = \mathtt{V}, \mathtt{read(A, J, V)}).$$

Similarly, the soundness of the application of rules RR2, WR1, and WR3 is derived by using (the contrapositive of) axiom A1, axiom A2, and axiom A3, respectively.
(1. *Soundness*) (ii). Suppose that $\mathtt{H :- k,G}$ is replaced by $\mathtt{H :- k_1,G}$ and $\mathtt{H :- k_2,G}$, by using WR3. Then $\mathtt{k}$ is the constraint $(\mathtt{c}, \mathtt{write(A,I,U,B)}, \mathtt{read(B,J,V)})$, $\mathtt{k_1}$ is the constraint $(\mathtt{c},\ \mathtt{I}=\mathtt{J},\ \mathtt{U}=\mathtt{V}, \mathtt{write(A,I,U,B)})$, and $\mathtt{k_2}$ is the constraint $(\mathtt{c},\ \mathtt{I}\neq\mathtt{J},\ \mathtt{write(A, I, U, B)},\ \mathtt{read(A, J, V)})$. By case split, we get:

$$\mathcal{A} \models \forall((\mathtt{c}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}) \leftrightarrow ((\mathtt{c}, \mathtt{I}=\mathtt{J}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}) \vee (\mathtt{c}, \mathtt{I}\neq\mathtt{J}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}))).$$

By using A3, we get:

$$\mathcal{A} \models \forall((\mathtt{c}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}) \leftrightarrow ((\mathtt{c}, \mathtt{I}=\mathtt{J}, \mathtt{U}=\mathtt{V}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}) \vee (\mathtt{c}, \mathtt{I}\neq\mathtt{J}, \mathtt{write(A, I, U, B)}, \mathtt{read(B, J, V)}, \mathtt{read(A, J, V)}))).$$

Since we have that $\mathcal{A} \models \forall(\mathtt{I}=\mathtt{J}, \mathtt{U}=\mathtt{V}, \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}) \rightarrow \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}))$, and we also have that $\mathcal{A} \models \forall(\mathtt{I} \neq \mathtt{J}, \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}), \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}) \rightarrow \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}))$, we get:

$$\mathcal{A} \models \forall((\mathtt{c}, \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}), \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})) \leftrightarrow ((\mathtt{c}, \mathtt{I}=\mathtt{J}, \mathtt{U}=\mathtt{V}, \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B})) \\ \vee\ (\mathtt{c}, \mathtt{I}\neq\mathtt{J}, \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}), \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})))).$$

(2. *Termination*) Let us define a relation, denoted $\prec$, on the set $vars(D)$ of the variables occurring in clause D as follows: $\mathtt{A} \prec \mathtt{B}$ iff the constraint $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B})$ occurs in D. The constraint $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B})$ denotes that the result of a write operation on array A is a new array B, and hence B is a variable not occurring as the fourth argument of any other write constraint. Thus, the transitive closure $\prec^+$ of $\prec$ is irreflexive, and since $vars(D)$ is a finite set, $\prec^+$ is a well-founded ordering on $vars(D)$. Note also that for every clause D' derived from D during the CONSTRAINT REPLACEMENT phase, we have that $vars(D') = vars(D)$.

Let us introduce the following measures for every clause E in the set S of clauses with variables in $vars(D)$:

(1) $\mu_n(E)$, which is the number of read constraints in the body of E,

(2) $\mu_r(E)$, which is the sum, for all constraints of the form $\mathtt{read}(\mathtt{B},\_,\_)$ in the body of E, of the number of variables A in $vars(D)$ such that $\mathtt{A} \prec^+ \mathtt{B}$,

(3) $\mu_p(E)$, which is the number of pairs $(\mathtt{I},\mathtt{J})$ of integer variables in $vars(D)$ such that $\mathtt{c} \not\sqsubseteq (\mathtt{I}\neq\mathtt{J})$, where c is the constraint in the body of E, and

(4) $\tau(E) =_{def} \langle\, \mu_n(E),\ \mu_r(E),\ \mu_p(E)\,\rangle$.

Now the termination of the CONSTRAINT REPLACEMENT phase is a consequence of the fact that if clause E is obtained from clause F by applying any of the rules in {RR1, RR2, WR1, WR2, WR3}, then $\tau(E) <_{lex} \tau(F)$, where $<_{lex}$ is the lexicographic ordering on triples of natural numbers (recall that $<_{lex}$ is a well-founded ordering). Indeed, the following facts hold:

(i) when applying rules RR1, WR1, and WR3(α), the measure μ_n decreases and no other rule application increases it,

(ii) when applying rule WR2, the measure μ_r decreases and no other rule application increases it (indeed, if clause E_1 is derived from clause E_2 by replacing a constraint $\mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$ by the new constraint $\mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$ with $\mathtt{A} \prec^+ \mathtt{B}$, then $\mu_r(E_1) < \mu_r(E_2)$), and

(iii) when applying rules RR2 and WR3(β), the measure μ_p decreases and no other rule application increases it.

(3. *Confluence*) In order to prove the confluence of the rewriting rules RR1, RR2, WR1, WR2, and WR3, since the constraint replacement is terminating (see Point 2), by the Newman Theorem [28] it is enough to prove *local confluence*, that is, it is enough to prove the following property: for all clauses C and all sets $S1$ and $S2$ of clauses, if $\{C\}$ can be rewritten *in one step* into the set $S1$ and also *in one step* into the set $S2$, then (i) $S1$ can be rewritten, in zero or more steps, into a set, say $\{C1_1, \ldots, C1_n\}$, of clauses, (ii) $S2$ can be rewritten, in zero or more steps, into a set, say $\{C2_1, \ldots, C2_n\}$, of clauses, and (iii) for $k = 1, \ldots, n$, clauses $C1_k$ and $C2_k$, after variable renaming, are of the form `H :- i1, c, G` and `H :- i2, c, G`, respectively, where: i1 and i2 are integer constraints, c is an array constraint, G is a conjunction of atoms, and $\mathbb{Z} \models \mathtt{i1} \leftrightarrow \mathtt{i2}$.

Now we consider the various cases for the one step rewritings. In what follows, for reason of conciseness we will write the prefix '1-', instead of 'one step', and when writing constraints, we allow ourselves to silently apply equivalences that hold in $\mathbb{Z}$.

• (*Case* RR1-RR1). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}), \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}), \mathtt{read}(\mathtt{A},\mathtt{K},\mathtt{W}))$ such that $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}=\mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{J}=\mathtt{K})$.

The constraint $\mathtt{k}$ 1-rewrites by RR1 into $\mathtt{k1} \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)}, \mathtt{read(A,K,W)})$, and $\mathtt{k}$ also 1-rewrites by RR1 into $\mathtt{k2} \equiv (\mathtt{c}, \mathtt{V}{=}\mathtt{W}, \mathtt{read(A,I,U)}, \mathtt{read(A,J,V)})$.

Now we have that $\mathtt{k1}$ can be 1-rewritten by RR1 into $\mathtt{k1}' \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{W}, \mathtt{read(A,I,U)})$, and $\mathtt{k2}$ can be 1-rewritten by RR1 into $\mathtt{k2}' \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{V}{=}\mathtt{W}, \mathtt{read(A,I,U)})$.

Since $\mathbb{Z} \models (\mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{W}) \leftrightarrow (\mathtt{U}{=}\mathtt{V}, \mathtt{V}{=}\mathtt{W})$, we get local confluence.

• (*Case* RR1-RR2). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{read(A,I,U)}, \mathtt{read(A,J,V)})$, such that $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I} \neq \mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$. Thus, in particular, $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$.

The constraint $\mathtt{k}$ 1-rewrites by RR1 into $\mathtt{k1} \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)})$, and $\mathtt{k}$ also 1-rewrites by RR2 into $\mathtt{k2} \equiv (\mathtt{c}, \mathtt{I} \neq \mathtt{J}, \mathtt{read(A,I,U)}, \mathtt{read(A,J,V)})$. Now, since $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$, $\mathtt{k2}$ 1-rewrites by RR1 into $\mathtt{k2}' \equiv (\mathtt{c}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)})$. Moreover, since $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$, we have that $\mathbb{Z} \models (\mathtt{c}{\downarrow}\mathbb{Z}, \mathtt{U}{=}\mathtt{V}) \leftrightarrow \mathtt{false}$, and $\mathbb{Z} \models (\mathtt{c}{\downarrow}\mathbb{Z}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}) \leftrightarrow \mathtt{false}$. Thus, we get local confluence.

• (*Case* RR1-WR1). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{read(A,I,U)}, \mathtt{read(A,J,V)}, \mathtt{write(A_0,I,U,A)})$, such that $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$.

The constraint $\mathtt{k}$ 1-rewrites by RR1 into $\mathtt{k1}_1 \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)}, \mathtt{write(A_0,I,U,A)})$, and $\mathtt{k}$ also 1-rewrites by WR1 into $\mathtt{k2}_1 \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)}, \mathtt{write(A_0,I,U,A)})$. Since $\mathtt{k1}_1$ and $\mathtt{k2}_1$ are syntactically equal, we get local confluence.

The constraint $\mathtt{k}$ may also 1-rewrite by WR1 into $\mathtt{k2}_2 \equiv (\mathtt{c}, \mathtt{read(A,J,V)}, \mathtt{write(A_0,I,U,A)})$. Now, $\mathtt{k1}_1$ is 1-rewritten by WR1 into $(\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{write(A_0,I,U,A)})$ and $\mathtt{k2}_2$ is 1-rewritten by WR1 into the same constraint $(\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{write(A_0,I,U,A)})$, and thus we get local confluence also in this case.

The constraint $\mathtt{k}$ also 1-rewrites by RR1 into $\mathtt{k1}_2 \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,J,V)}, \mathtt{write(A_0,I,U,A)})$. Now, since the constraints $\mathtt{k1}_2$, $\mathtt{k2}_1$, and $\mathtt{k2}_2$ can all be 1-rewritten by RW1 into a constraint of the form $(\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{write(A_0,I,U,A)})$, we get local confluence.

• (*Case* RR1-WR2) and (*Case* RR1-WR3). These cases are impossible because: (i) it is not the case that $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I} \neq \mathtt{J})$ hold, and (ii) it is not the case that $\mathtt{k}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{k}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}{=}\mathtt{J})$ hold.

Now we have to consider all other cases of the 1-rewritings when the first rule is not RR1. The proofs of all these cases are similar to ones shown above, and we leave them to the reader.

Here we only show the following cases.

• (*Case* RR2-WR1). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{d}, \mathtt{read(A,I,U)}, \mathtt{read(A,J,V)}, \mathtt{write(A_0,I,U,A)})$, where $\mathtt{d}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I} \neq \mathtt{J})$ and $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$ and $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$.

The constraint $\mathtt{k}$ 1-rewrites by RR2 into $\mathtt{k1} \equiv (\mathtt{k}, \mathtt{I} \neq \mathtt{J})$. The constraint $\mathtt{k}$ also 1-rewrites by WR1 into $\mathtt{k2}_1 \equiv (\mathtt{d}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)}, \mathtt{write(A_0,I,U,A)})$. Now, since $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$, $\mathtt{k1}$ 1-rewrites by RR1 into $\mathtt{k1}' \equiv (\mathtt{d}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}, \mathtt{read(A,I,U)}, \mathtt{write(A_0,I,U,A)})$. Moreover, since $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$, we have that $\mathbb{Z} \models (\mathtt{d}{\downarrow}\mathbb{Z}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}) \leftrightarrow \mathtt{false}$, and $\mathbb{Z} \models (\mathtt{d}{\downarrow}\mathbb{Z}, \mathtt{U}{=}\mathtt{V}) \leftrightarrow \mathtt{false}$. Thus, we get local confluence.

The constraint $\mathtt{k}$ also 1-rewrites by WR1 into $\mathtt{k2}_2 \equiv (\mathtt{d}, \mathtt{U}{=}\mathtt{U}, \mathtt{read(A,J,V)}, \mathtt{write(A_0,I,U,A)})$. This constraint by WR1 is 1-rewritten into $(\mathtt{d}, \mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{U}, \mathtt{write(A_0,I,U,A)})$. Now $\mathtt{k1}'$ is 1-rewritten by WR1 into $(\mathtt{d}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{U}, \mathtt{write(A_0,I,U,A)})$. Since $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{U} \neq \mathtt{V})$ and $\mathtt{d}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$, we have that $\mathbb{Z} \models (\mathtt{d}{\downarrow}\mathbb{Z}, \mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{U}) \leftrightarrow \mathtt{false}$ and $\mathbb{Z} \models (\mathtt{d}{\downarrow}\mathbb{Z}, \mathtt{I} \neq \mathtt{J}, \mathtt{U}{=}\mathtt{V}, \mathtt{U}{=}\mathtt{U}) \leftrightarrow \mathtt{false}$. Thus, we get local confluence.

• (*Case* WR1-WR1). (Case of overlapping redexes on $\mathtt{write}$). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{write(A,I,U,B)}, \mathtt{read(B,J,V)}, \mathtt{read(B,K,W)})$, where $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}{=}\mathtt{K})$.

The constraint $\mathtt{k}$ 1-rewrites by WR1 into $\mathtt{k1} \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{V}, \mathtt{write(A,I,U,B)}, \mathtt{read(B,K,W)})$. Also the constraint $\mathtt{k}$ also 1-rewrites by WR1 into $\mathtt{k2} \equiv (\mathtt{c}, \mathtt{U}{=}\mathtt{W}, \mathtt{write(A,I,U,B)}, \mathtt{read(B,J,V)})$.

Now $\mathtt{k1}$ can be 1-rewritten by WR1 into $\mathtt{k1}' \equiv (\mathtt{c}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}))$ and $\mathtt{k2}$ can be 1-rewritten by WR1 into $\mathtt{k2}' \equiv (\mathtt{c}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}))$. By commutativity of constraints in $\mathbb{Z}$, we get local confluence.

• (*Case* WR1-WR1). (Case of overlapping redexes on `read`). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}), \mathtt{write}(\mathtt{A}, \mathtt{K}, \mathtt{W}, \mathtt{B}))$, where $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{I}\!=\!\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \sqsubseteq (\mathtt{K}\!=\!\mathtt{J})$. This case is impossible because there are two `write(A,K,W,B)` constraints with the same fourth argument.

• (*Case* WR3-WR3). (Case of overlapping redexes on `write`). Let us consider the constraint $\mathtt{k} \equiv (\mathtt{c}, \mathtt{write}(\mathtt{A{,}I{,}U{,}B}), \mathtt{read}(\mathtt{B{,}J{,}V}), \mathtt{read}(\mathtt{B{,}K{,}W}))$, where $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!=\!\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!\neq\!\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!=\!\mathtt{K})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!\neq\!\mathtt{K})$.

By considering the constraint $(\mathtt{c}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}))$, $\mathtt{k}$ 1-rewrites by WR3 into the two constraints:

$\mathtt{k1}_\alpha \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{J}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{B}, \mathtt{K}, \mathtt{W}))$
$\mathtt{k1}_\beta \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{J}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}), \mathtt{read}(\mathtt{B}, \mathtt{K}, \mathtt{W}))$.

By considering the constraint $(\mathtt{c}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{B}, \mathtt{K}, \mathtt{W}))$, $\mathtt{k}$ also 1-rewrites by WR3 into the two constraints:

$\mathtt{k2}_\alpha \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{K}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}))$
$\mathtt{k2}_\beta \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{K}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{W}), \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}))$.

From $\mathtt{k1}_\alpha$ and $\mathtt{k1}_\beta$, since $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!=\!\mathtt{K})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!\neq\!\mathtt{K})$, we get by 1-rewritings by WR3 the following constraints:

$\mathtt{k1}_{\alpha\alpha} \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{J}, \mathtt{I}\!=\!\mathtt{K}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}))$
$\mathtt{k1}_{\alpha\beta} \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{J}, \mathtt{I}\!\neq\!\mathtt{K}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{W}))$
$\mathtt{k1}_{\beta\alpha} \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{J}, \mathtt{I}\!=\!\mathtt{K}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}))$
$\mathtt{k1}_{\beta\beta} \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{J}, \mathtt{I}\!\neq\!\mathtt{K}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}), \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{W}))$.

From $\mathtt{k2}_\alpha$ and $\mathtt{k2}_\beta$, since $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!=\!\mathtt{J})$ and $\mathtt{c}{\downarrow}\mathbb{Z} \not\sqsubseteq (\mathtt{I}\!\neq\!\mathtt{J})$, we get by 1-rewritings by WR3 the following constraints:

$\mathtt{k2}_{\alpha\alpha} \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{K}, \mathtt{I}\!=\!\mathtt{J}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}))$
$\mathtt{k2}_{\alpha\beta} \equiv (\mathtt{c}, \mathtt{I}\!=\!\mathtt{K}, \mathtt{I}\!\neq\!\mathtt{J}, \mathtt{U}\!=\!\mathtt{W}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}))$
$\mathtt{k2}_{\beta\alpha} \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{K}, \mathtt{I}\!=\!\mathtt{J}, \mathtt{U}\!=\!\mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{W}))$
$\mathtt{k2}_{\beta\beta} \equiv (\mathtt{c}, \mathtt{I}\!\neq\!\mathtt{K}, \mathtt{I}\!\neq\!\mathtt{J}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}), \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{W}), \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}))$.

Now let us consider the pair $(\mathtt{k1}_{\alpha\alpha}, \mathtt{k2}_{\alpha\alpha})$ of constraints. We have that: $\mathbb{Z} \models \mathtt{k1}_{\alpha\alpha}{\downarrow}\mathbb{Z} \leftrightarrow \mathtt{k2}_{\alpha\alpha}{\downarrow}\mathbb{Z}$. Actually, $\mathtt{k1}_{\alpha\alpha}$ and $\mathtt{k2}_{\alpha\alpha}$ are syntactically equal, modulo commutativity of conjunction. The same holds for the other pairs of constraints: $(\mathtt{k1}_{\alpha\beta}, \mathtt{k2}_{\beta\alpha})$, $(\mathtt{k1}_{\beta\alpha}, \mathtt{k2}_{\alpha\beta})$, and $(\mathtt{k1}_{\beta\beta}, \mathtt{k2}_{\beta\beta})$. Thus, we get local confluence.

(*Case* WR3-WR3). (Case of overlapping redexes on `read`). As for the case WR1-WR1, this case is impossible. □

Let us continue our verification of the *SeqInit* program by performing the CONSTRAINT REPLACEMENT transformation phase. First, we simplify clause 8 by replacing the integer constraint in its body with an equivalent one. We get:

8r. $\mathtt{incorrect}$:- $\mathtt{K}\!=\!\mathtt{J}\!+\!1,\ \mathtt{J}\!\geq\!0,\ \mathtt{K}\!\leq\!\mathtt{H},\ \mathtt{G}\!=\!\mathtt{H}\!-\!1,\ \mathtt{N}\!=\!\mathtt{H}\!+\!1,\ \mathtt{Z}\!=\!\mathtt{W}\!+\!1,\ \mathtt{U}\!\geq\!\mathtt{V},$
$\mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{K}, \mathtt{V}),\ \mathtt{read}(\mathtt{B}, \mathtt{G}, \mathtt{W}),\ \mathtt{write}(\mathtt{B}, \mathtt{H}, \mathtt{Z}, \mathtt{A}),\ \mathtt{p}(\mathtt{H}, \mathtt{N}, \mathtt{B}).$

Since $\mathtt{J}\!\neq\!\mathtt{H}$ is entailed by the constraint in clause 8r, we apply rule WR2 and we replace '$\mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{U})$, $\mathtt{write}(\mathtt{B}, \mathtt{H}, \mathtt{Z}, \mathtt{A})$' by '$\mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{U}), \mathtt{write}(\mathtt{B}, \mathtt{H}, \mathtt{Z}, \mathtt{A})$'. We get:

8r.1 `incorrect :- K=J+1, J≥0, K≤H, G=H−1, N=H+1, Z=W+1, U≥V,`
`read(B,J,U),` $\underline{\texttt{read(A,K,V)}}$`, read(B,G,W),` $\underline{\texttt{write(B,H,Z,A)}}$`, p(H,N,B).`

Then, since neither `K=H` nor `K≠H` is entailed by the constraint in clause 8r.1, we apply rule WR3 and we obtain the following two clauses (we have underlined the constraints involved in this replacement):

8r.2 `incorrect :- K=J+1, J≥0, K≤H, G=H−1, N=H+1, Z=W+1, U≥V,`
$\underline{\texttt{K=H}}$, $\underline{\texttt{Z=V}}$`, read(B,J,U), read(B,G,W),` $\underline{\texttt{write(B,H,Z,A)}}$`, p(H,N,B).`

8r.3 `incorrect :- K=J+1, J≥0, K≤H, G=H−1, N=H+1, Z=W+1, U≥V,`
$\underline{\texttt{K≠H}}$`, read(B,J,U),` $\underline{\texttt{read(B,K,V)}}$`, read(B,G,W),` $\underline{\texttt{write(B,H,Z,A)}}$`, p(H,N,B).`

Finally, since `J=G` is entailed by the constraint in clause 8r.2 (indeed, `K=J+1,G=H−1,K=H`), we apply rule RR1 to clause 8r.2 and we replace the constraint `read(B,G,W)` by the constraint `W=U`, thereby deriving the unsatisfiable constraint '`W=U,Z=W+1,Z=V,U≥V`'. Thus, clause 8r.2 is removed by the subsequent CLAUSE REMOVAL phase. From clause 8r.3, by rewriting '`K≤H, K ≠ H`' as '`K≤H−1`', we get:

9. `incorrect :- K=J+1, J≥0, K≤H−1, G=H−1, N=H+1, Z=W+1, U≥V,`
`read(B,J,U), read(B,K,V), read(B,G,W), write(B,H,Z,A), p(H,N,B).`

5.3. Definition and Folding

The DEFINITION & FOLDING phase introduces new predicate definitions by suitable generalizations of the constraints. Generalization guarantees the termination of *VCTransf*. In particular, by using the Generalization Algorithm presented in Figure 2, we enforce the introduction of a *finite* set *Defs* of new predicate definitions such that all clauses derived by applying unfolding, constraint replacement, and clause removal to the clauses in *Defs* can be folded by using clauses in the set *Defs* itself.

Unfortunately, in some cases our generalization technique may introduce an overly general new predicate definition whose unfolding may generate constrained facts, thereby preventing us to prove that `incorrect` $\notin M(P)$, even if the given array manipulating program is correct. Informally, this case may happen when the new predicate provides a too coarse overapproximation of the set of configurations that are reachable in a backward way from the error configurations. Indeed, this overapproximation includes initial configurations (because the predicate has constrained facts), which are not actually reachable from the error configurations (because the program is correct). Clearly, due to the undecidability of program correctness, no generalization technique can guarantee termination and, at the same time, the derivation of a program without constrained facts whenever `incorrect` $\notin M(P)$.

The DEFINITION & FOLDING phase works as follows. Let $C1$ in $R(C)$ be a clause of the form `H :- c, p(X)`. In order to reason about the predicate definitions introduced in previous steps of *VCTransf*, we structure the set *Defs* as a tree of clauses, where clause A is the parent of clause B if B has been introduced for folding a clause in $R(A)$. If in *Defs* there is (a variant of) a clause D: `newp(X) :- d, p(X)` such that *vars*(`d`) $\subseteq$ *vars*(`c`) and `c` $\sqsubseteq$ `d`, then we fold $C1$ using D. Otherwise, we introduce a clause of the form `newp(X) :- gen, p(X)` where: (i) `newp` is a predicate symbol occurring neither in the initial program nor in *Defs*, and (ii) `gen` is a constraint such that *vars*(`gen`) $\subseteq$ *vars*(`c`) and `c` $\sqsubseteq$ `gen`. The constraint `gen` is called a *generalization* of the constraint `c`.

Many different generalizations of constraints can be defined. In Figure 2, we propose a Generalization Algorithm for computing one such generalization. This algorithm is parametric with respect to the operator $\ominus$ that is used for generalizing linear constraints. $\ominus$ is a binary operator such that, for any two integer constraints $\mathtt{i}_1$ and $\mathtt{i}_2$, we have $\mathit{vars}(\mathtt{i}_1 \ominus \mathtt{i}_2) \subseteq \mathit{vars}(\mathtt{i}_2)$ and $\mathtt{i}_2 \sqsubseteq \mathtt{i}_1 \ominus \mathtt{i}_2$. We refer to [9, 17, 43] for the definition of generalization operators for linear constraints based on *widening* and *convex hull*.

Input: (i) A clause C, (ii) a clause in $R(C)$ of the form `H :- c, p(X)`, and (iii) a tree *Defs* of predicate definitions.
Output: A constraint `gen` which is a *generalization* of the constraint `c`.

Let `c` be of the form $\mathtt{i}_1, \mathtt{rw}_1$, where $\mathtt{i}_1$ is an integer constraint and $\mathtt{rw}_1$ is a conjunction of `read` and `write` constraints. Without loss of generality, we assume that all occurrences of integers in `read` constraints of `c` are distinct variables not occurring in `X` (this condition can always be fulfilled by adding extra integer equalities).

1. Delete all `write` constraints from $\mathtt{rw}_1$, hence deriving $\mathtt{r}_1$.
2. Compute the projection $\mathtt{i}_2$ (in the rationals $\mathbb{Q}$) of the constraint $\mathtt{i}_1$ onto $\mathit{vars}(\mathtt{r}_1) \cup \{\mathtt{X}\}$. (Recall that the projection in $\mathbb{Q}$ of a constraint $\mathtt{c}(\mathtt{Y}, \mathtt{Z})$ onto the tuple `Y` of variables is a constraint $\mathtt{c}_\mathtt{p}(\mathtt{Y})$ such that $\mathbb{Q} \models \forall\, \mathtt{Y}(\mathtt{c}_\mathtt{p}(\mathtt{Y}) \leftrightarrow \exists\, \mathtt{Z}\, \mathtt{c}(\mathtt{Y}, \mathtt{Z}))$.)
3. Delete from $\mathtt{r}_1$ all `read(A, I, V)` constraints such that either (i) `A` does not occur in `X`, or (ii) `V` does not occur in $\mathtt{i}_2$, thereby deriving a new value for $\mathtt{r}_1$. If at least one `read` has been deleted during this step, then go to Step 2.
4. Let $\mathtt{i}_2, \mathtt{r}_2$ be the constraint obtained after the possibly repeated executions of Steps 2–3.
   If in *Defs* there is an ancestor (defined as the reflexive, transitive closure of the parent relation) of C of the form $\mathtt{H}_0$:- $\mathtt{i}_0, \mathtt{r}_0, \mathtt{p}(\mathtt{X})$ such that $\mathtt{r}_0, \mathtt{p}(\mathtt{X})$ is a subconjunction of $\mathtt{r}_2, \mathtt{p}(\mathtt{X})$,
   then let g be $\mathtt{i}_0 \ominus \mathtt{i}_2$. Define the constraint `gen` as $\mathtt{g}, \mathtt{r}_0$;
   else define the constraint `gen` as $\mathtt{i}_2, \mathtt{r}_2$.

Figure 2. The Generalization Algorithm.

Let us make some remarks on the Generalization Algorithm. Step 1 is justified by the fact that `write` constraints are redundant after the application of the *read-over-write* constraint replacements RW1–RW3. After Step 1 we have $\mathit{vars}(\mathtt{i}_1, \mathtt{r}_1) \subseteq \mathit{vars}(\mathtt{c})$ and $\mathtt{i}_1, \mathtt{rw}_1 \sqsubseteq \mathtt{i}_1, \mathtt{r}_1$.

At Step 2 we compute the projection $\mathtt{i}_2$ of $\mathtt{i}_1$ *in the rationals* $\mathbb{Q}$ (and hence $\mathtt{i}_1 \sqsubseteq \mathtt{i}_2$ holds in the domain of the integers), because linear constraints are not closed under projection in the domain of the integers. We have that $\mathit{vars}(\mathtt{i}_2, \mathtt{r}_1) \subseteq \mathit{vars}(\mathtt{i}_1, \mathtt{r}_1)$ and $\mathtt{i}_1, \mathtt{r}_1 \sqsubseteq \mathtt{i}_2, \mathtt{r}_1$.

At Step 3 the deletion of constraints of the form `read(A, I, V)`, where `A` does not occur in `X`, is motivated by the fact that `A` can be treated as an existentially quantified variable, and $\exists\mathtt{A}.\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$ holds for all `I` and `V`. The deletion of constraints of the form `read(A, I, V)`, where `V` does not occur in $\mathtt{i}_2$, is motivated by the fact that `V` can be treated as an existentially quantified variable and, since by construction $\mathtt{i}_2$ ensures that the index `I` is within bounds, we have that $\exists\mathtt{V}.\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$ holds for all `A` and `I`. Thus, at the end of Steps 2–3, $\mathit{vars}(\mathtt{i}_2, \mathtt{r}_2) \subseteq \mathit{vars}(\mathtt{i}_2, \mathtt{r}_1)$ and $\mathtt{i}_2, \mathtt{r}_1 \sqsubseteq \mathtt{i}_2, \mathtt{r}_2$.

Step 4 computes a generalization `g` of the integer constraint $\mathtt{i}_2$ if an ancestor clause in *Defs* contains a subconjunction $\mathtt{r}_0$ of the `read` constraint $\mathtt{r}_2$. We will show in the next section that this condition guarantees the termination of the *VCTransf* strategy. If the condition of the *If-then-else* holds, then $\mathit{vars}(\mathtt{gen}) = \mathit{vars}(\mathtt{g}, \mathtt{r}_0) \subseteq \mathit{vars}(\mathtt{i}_2, \mathtt{r}_2)$ and $\mathtt{i}_2, \mathtt{r}_2 \sqsubseteq \mathtt{g}, \mathtt{r}_0 = \mathtt{gen}$. If the condition of the *If-then-else* does not hold, then `gen` is $\mathtt{i}_2, \mathtt{r}_2$, and hence $\mathit{vars}(\mathtt{gen}) = \mathit{vars}(\mathtt{i}_2, \mathtt{r}_2)$. Thus, after Step 4, $\mathit{vars}(\mathtt{gen}) \subseteq \mathit{vars}(\mathtt{c})$ and $\mathtt{c} \sqsubseteq \mathtt{gen}$.

All generalization operators $\ominus$ used at Step 4 guarantee the termination and soundness of *VCTransf*, but they may have an influence on the number of transformation steps needed to terminate, and also on the success of the verification (recall that *VCTransf* may terminate without proving or disproving correctness). The comparison among the various generalization operators we have considered, has been done on an experimental basis and the results of that comparison are reported in Section 6.

Let us continue our program verification example by performing, starting from clause 9, the DEFINITION & FOLDING phase of the *VCTransf* strategy. In order to fold clause 9, we will introduce a new predicate definition by applying the Generalization Algorithm. We start off by renaming the variables occurring in clause 9. This renaming has the objective of simplifying the matching process of Step 4 of the Generalization Algorithm. We get the following clause:

9r. `incorrect :- K=J+1, J≥0, K≤I−1, G=I−1, N=I+1, Z=W+1, U≥V,`
`read(A,J,U), read(A,K,V), read(A,G,W), write(A,I,Z,A1), p(I,N,A).`

Now, we delete the `write` constraint (Step 1) and we project the integer constraints (Step 2), thereby deleting `Z=W+1`. We get a constraint where the variable `W` occurs in `read(A,G,W)` only. Thus, after deleting the constraint `read(A,G,W)` (Step 3) and by applying projection again (this step results in the deletion of `G=I−1`), we derive the constraint:

`K=J+1, J≥0, K≤I−1, N=I+1, U≥V, read(A,J,U), read(A,K,V).`

Finally, we apply Step 4 of the Generalization Algorithm and, by using the convex hull operator, we compute a generalization of the integer constraint `K=J+1, J≥0, J≤N−2, N≤I, U≥V` occurring in the body of clause 5, and the constraint `K=J+1, J≥0, K≤I−1, N=I+1, U≥V` obtained after Steps 1–3. We get the following new predicate definition:

10. `new1(I,N,A) :- K=J+1, J≥0, J≤N−2, J≤I−2, N≤I+1, U≥V,`
`read(A,J,U), read(A,K,V), p(I,N,A).`

By folding clause 9r using clause 10, we get:

11. `incorrect :- K=J+1, J≥0, K≤I−1, G=I−1, N=I+1, Z=W+1, U≥V,`
`read(A,J,U), read(A,K,V), read(A,G,W), write(A,I,Z,A1), new1(I,N,A).`

5.4. Termination and Soundness of the *VCTransf* Transformation Strategy

The following theorem, together with Theorem 4.1, ensures that our verification method, consisting of two steps *VCGen* and *VCTransf*, terminates and is sound.

Theorem 5.2. (*Termination and Soundness of VCTransf*) (i) The *VCTransf* strategy terminates.
(ii) Let program T be the output of the *VCTransf* strategy applied on the input program VC. Then, `incorrect` $\in M(VC)$ iff `incorrect` $\in M(T)$.

Proof:
(i) The *VCTransf* strategy is parametric with respect to the generalization operator $\ominus$ on integer constraints used in the Generalization Algorithm. We assume that $\ominus$ has a property ensuring that only finite chains of generalizations of any given integer constraint can be generated by applying the operator. This assumption is formalized by the following property:

(F) *if* $\langle g_0, g_1, \ldots \rangle$ is an infinite sequence of integer constraints and, for all $m > 0$, there exist an index $j < m$ and an integer constraint i such that $g_m = g_j \ominus i$,
then there exist k, n such that $k < n$ and $g_n \sqsubseteq g_k$.

The already mentioned generalization operators presented in [9, 17, 43] satisfy property (F).

Let us first note that the UNFOLDING, CONSTRAINT REPLACEMENT, CLAUSE REMOVAL, and DEFINITION & FOLDING phases terminate. In particular, constraint satisfiability and entailment are decidable for the class of quantifier-free array constraints we are considering, and hence can be checked by a terminating solver (note that completeness of the solver is not necessary for the termination of the *VCTransf* strategy), and each sequence of constraint replacements terminates (see Theorem 5.1).

Next we note that the while-loop of the *VCTransf* strategy terminates if and only if the set of new predicate definitions that, during the execution of the strategy, is introduced by executions of the DEFINITION & FOLDING phase is finite. Indeed, each new predicate definition is added to *InDefs* and processed in one execution of the body of the while-loop.

Let us now prove that the set of new predicate definitions is finite.
By construction, each predicate definition is of the form $\mathtt{newp(X)}$:- $\mathtt{i, r, p(X)}$, where: (1) $\mathtt{i}$ is an integer constraint, (2) $\mathtt{r}$ is a conjunction of array constraints of the form $\mathtt{read(A, I, V)}$, where $\mathtt{A}$ is a variable in $\mathtt{X}$ and the variables $\mathtt{I}$ and $\mathtt{V}$ occur in $\mathtt{i}$ only (see Step 1 of the Generalization Algorithm), and (3) $\mathtt{p(X)}$ is a predicate occurring in VC.

The proof proceeds by contradiction. Let assume that the set of new predicate definitions is infinite, and hence there exists an infinite sequence $\langle D_0, D_1, \ldots\rangle$ of clauses in *Defs* such that, for $i \geq 0$, D_i is the parent of D_{i+1}. Since the *else* branch of the *If-then-else* of Step 4 of the Generalization Algorithm can only be applied a finite number of consecutive times during the construction of the sequence $\langle D_0, D_1, \ldots\rangle$, we can extract from that sequence an infinite subsequence of clauses of the form:

$\langle$ $\mathtt{newp_0(X)}$:- $\mathtt{g_0, r_0, p(X)}$, $\mathtt{newp_1(X)}$:- $\mathtt{g_1, r_0, p(X)}$, $\mathtt{newp_2(X)}$:- $\mathtt{g_2, r_0, p(X)}$, $\ldots$ $\rangle$

where, for $\mathtt{m} = 1, 2, \ldots$, $\mathtt{g_m} = \mathtt{g_j} \ominus \mathtt{i}$, for some $\mathtt{j} < \mathtt{m}$ and integer constraint $\mathtt{i}$. By Property (F) we get that there exist $\mathtt{k, n}$ such that $\mathtt{k} < \mathtt{n}$ and $\mathtt{g_n} \sqsubseteq \mathtt{g_k}$. Thus, we have reached a contradiction. Indeed, according to the DEFINITION & FOLDING phase, the clause $D_{\mathtt{n}}$: $\mathtt{newp_n(X)}$:- $\mathtt{g_n, r_0, p(X)}$ should have not been introduced because in *Defs* there is a clause $D_{\mathtt{k}}$: $\mathtt{newp_k(X)}$:- $\mathtt{g_k, r_0, p(X)}$ such that $\mathtt{g_n} \sqsubseteq \mathtt{g_k}$, and any clause that can be folded using $D_{\mathtt{n}}$ could have been folded using $D_{\mathtt{k}}$.

Thus, the set of new predicate definitions is finite and the *VCTransf* strategy terminates.

(ii) Each transformation step in the *VCTransf* strategy is a sound application of the rules presented in Section 3. In particular, by Theorem 5.1, each constraint replacement in the CONSTRAINT REPLACEMENT phase is a sound application of the constraint replacement rule. Moreover, every clause defining a new predicate introduced during the DEFINITION & FOLDING phase is unfolded once during the execution of the strategy. Thus, the soundness of the strategy with respect to the least $\mathcal{A}$-model semantics follows from Theorem 3.1. □

Let us now conclude the verification of the *SeqInit* program. The *VCTransf* strategy proceeds by performing a second iteration of the body of the while-loop because *InDefs* is not empty (indeed, at this point clause 10 belongs to *InDefs*).

UNFOLDING. By unfolding clause 10 we get the following clause:

```
12. new1(I,N,A) :- K=J+1, J≥0, J≤N-2, J≤I-2, N≤I+1, U≥V,
              1≤H, H≤N-1, G=H-1, I=H+1, Z=W+1,
              read(A,J,U), read(A,K,V), read(B,G,W), write(B,H,Z,A), p(H,N,B).
```

CONSTRAINT REPLACEMENT. Then, by simplifying the integer constraints and applying rules RR1, WR2, and WR3, from clause 12 we get the following clause:

```
13. new1(I,N,A) :- K=J+1, I=H+1, Z=W+1, G=H−1, N≤H+2,
            K≤H−1, K≥1, N≥H+1, U≥V,
            read(B,J,U), read(B,K,V), read(B,G,W), write(B,H,Z,A), p(H,N,B).
```

DEFINITION & FOLDING. In order to fold clause 13 we introduce the following clause, whose body is derived by computing the widening [7, 9] of the integer constraints in the ancestor clause 10 with respect to the integer constraints in (a renamed version of) clause 13 (recall that the widening of a constraint `c` with respect to a constraint `d` is the conjunction of all atomic constraints of `c` that are entailed by `d`):

```
14. new2(I,N,A) :- K=J+1, J≥0, J≤I−2, J≤N−2, U≥V,
            read(A,J,U), read(A,K,V), p(I,N,A).
```

By folding clause 13 using clause 14, we get:

```
15. new1(I,N,A) :- K=J+1,I=H+1,Z=W+1,G=H−1,N≤H+2,K≤H−1,K≥1,N≥H+1,U≥V,
            read(B,J,U), read(B,K,V), read(B,G,W), write(B,H,Z,A), new2(H,N,B).
```

Now we perform the third iteration of the body of the while-loop of the strategy starting from the newly introduced definition, that is, clause 14. After some executions of the UNFOLDING and CONSTRAINT REPLACEMENT phases, followed by a final FOLDING phase, from clause 14 we get:

```
16. new2(I,N,A) :- K=J+1, I=H+1, Z=W+1, G=H−1, K≤H−1, K≥1, N≥H+1, U≥V,
            read(B,J,U), read(B,K,V), read(B,G,W), write(B,H,Z,A), new2(H,N,B).
```

The transformed program is made out of clauses 11, 15, and 16. Since this program has no constrained facts, by executing the REMOVAL OF USELESS CLAUSES phase, we derive the empty program T, and we conclude that `incorrect` $\notin M(T)$ and the Hoare triple $\{\varphi_{init}\}$ *SeqInit* $\{\neg\varphi_{error}\}$ is valid.

6. Experimental Evaluation

We have performed an experimental evaluation of our method on a benchmark set consisting of programs manipulating arrays. In order to evaluate our method, we have implemented the transformation strategies *VCGen* and *VCTransf* of Sections 4 and 5, respectively, as modules of the VeriMAP software model checker [11]. The VeriMAP tool consists of: (i) a front-end module, based on a custom implementation of the C Intermediate Language (CIL) visitor pattern [42], which translates a C program, together with its precondition and postcondition, into a set of CLP(Array) facts, and (ii) a back-end module, implemented in Prolog, for CLP(Array) program transformation that generates the verification conditions and applies the *VCTransf* strategy. The back-end also includes a solver for quantifier free formulas of the theory of arrays that checks satisfiability and entailment for array constraints by using the rules RR1–WR3 (see Section 5.2) and the solver for linear equalities and inequalities over the rationals provided by the `clpq` library of SICStus Prolog.

We have compared our results with those obtained by the state-of-the-art verifiers BOOSTER [3] and SMACK+Corral [26] (SMACK, for short). The results of our experiments, summarized in Tables 2 and 3, show that our approach is quite effective and efficient in practice.

Now we briefly discuss the programs, mostly taken from the literature [2, 5, 8, 13, 25, 36, 49], that have been considered in our experimental evaluation. The source code of these programs can be found in `http://map.uniroma2.it/smc/arrays/`. Every program verification experiment we have performed, consisted in checking the validity of a triple of the form $\{true\}$ *prog* $\{\neg\,\varphi_{error}\}$, where *prog* and $\neg\,\varphi_{error}$ are given in Table 1. The validity check was done by using either VeriMAP, or BOOSTER, or SMACK.

Precondition: *true* Example: *prog*	Postcondition to be verified: $\neg\, \varphi_{error}$
1. *bubblesort-inner*	$\forall k.\ (0 \leq i < n \ \wedge\ 0 \leq k < j \ \wedge\ j = n-i-1) \rightarrow a[k] \leq a[j]$
2. *bubblesort*	$\forall i, j.\ (0 \leq i < j \wedge j < n) \rightarrow a[i] \leq a[j]$
3. *insertionsort-inner*	$\forall k.\ (0 \leq i < n \wedge j+1 < k \leq i) \rightarrow a[k] > x$
4. *selectionsort-inner*	$\forall k.\ (0 \leq i \leq k < n) \rightarrow a[k] \geq a[i]$
5. *copy*	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = b[i]$
6. *copy-partial*	$\forall i.\ (0 \leq i < k \leq n) \rightarrow a[i] = b[i]$
7. *copy-reverse*	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = b[n-i-1]$
8. *difference*	$\forall i.\ (0 \leq i < n) \rightarrow c[i] = a[i] - b[i]$
9. *sum*	$\forall i.\ (0 \leq i < n) \rightarrow c[i] = a[i] + b[i]$
10. *find-first-non-null-1*	$(0 \leq p < n) \rightarrow a[p] \neq 0$
11. *find-first-non-null-2*	$(0 \leq p < n) \rightarrow (a[p] \neq 0 \ \wedge\ (\forall i.\ (0 \leq i < p) \rightarrow a[i] = 0))$
12. *find*	$(0 \leq p < n) \rightarrow a[p] = x$
13. *init-constant*	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = d$
14. *init-partial-zero*	$\forall i.\ (0 \leq i < k \leq n) \rightarrow a[i] = 0$
15. *init-backward-zero*	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = 0$
16. *init-non-constant*	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = 2\,i + d$
17. *init-sequence*	$\forall i.\ (1 \leq i < n) \rightarrow a[i] = a[i-1] + 1$
18. *max*	$\forall i.\ (0 \leq i < n) \rightarrow m \geq a[i]$
19. *partition*	$(\forall i.\ (0 \leq i < j) \rightarrow b[i] \geq 0) \ \wedge\ (\forall i.\ (0 \leq i < k) \rightarrow c[i] < 0)$
20. *rearrange-in-situ*	$(\forall k.\ (0 \leq k < i) \rightarrow a[k] \geq 0) \ \wedge\ (\forall k.\ (j < k < n) \rightarrow a[k] < 0)$

Table 1. Array programs and postconditions. The arrays a, b, and c are assumed to have dimension n.

Programs *bubblesort-inner*, *insertionsort-inner*, and *selectionsort-inner* are the inner loops of the standard textbook versions of those sorting algorithms. Program *bubblesort* is the bubblesort algorithm of the benchmark suite of BOOSTER. Programs *copy* and *copy-partial* perform the element-wise copy of the entire input array or a portion of it, respectively. Program *copy-reverse* copies the input array in reverse order, by making use of a temporary extra copy. Programs *difference* and *sum* perform the element-wise difference and sum, respectively, of two input arrays. Programs *find-first-non-null-1* and *find-first-non-null-2* both return the position p of the first non-zero element of the input array by using two different algorithms. Program *find* returns the position p of the first occurrence of a given value x in the input array. Program *init-constant* initializes to the integer d all elements of the input array. Program *init-partial-zero* initializes to 0 a portion of the input array (the initialization starts from the first element). Program *init-backward-zero* initializes to 0 the entire input array (the initialization starts from the last element). Programs *init-non-constant* and *init-sequence* initialize the input array using values that depend on the element position or the preceding element, respectively. Program *max* computes the maximum element of the input array. Program *partition* copies the non-negative and negative elements of the input array into two distinct arrays. Program *rearrange-in-situ*, rearranges the elements of the input array, so that all negative elements are placed to the right of the non-negative ones.

In order to verify the above programs, we have applied the *VCTransf* strategy using different generalization operators for linear constraints. In particular, when computing new predicate definitions using the Generalization Algorithm, we have considered the *Gen*$_W$ operator, which performs widening, and the *Gen*$_{CHW}$ operator, which in the same generalization step performs widening and convex hull. We have also combined these operators with a *delay* mechanism, thereby obtaining *Gen*$_{WD}$ and *Gen*$_{CHWD}$, respectively. The *Gen*$_{WD}$ operator applies the *Gen*$_W$ operator and the convex hull operator in an alternate way, and *Gen*$_{CHWD}$ does the same for *Gen*$_{CHW}$. The interested reader may refer to [12, 17] for details on these operators.

In Table 2 we report the results of our experimental evaluation obtained by using the VeriMAP tool with the four generalization operators mentioned above and the BOOSTER tool. All programs are assumed to manipulate arrays of unknown dimension n. The SMACK tool has not been considered in this evaluation because it can only deal with arrays of known dimension. For each program that has been proved correct, we report the time (in seconds) taken to verify the postcondition of interest. In Table 2 the entry '*unknown*' means that the tool terminates without being able to prove or disprove the postcondition, while the entry '*timeout*' means that the tool did not provide an answer within 300 seconds. At the bottom of Table 2 we also report: (i) the *precision*, that is, the ratio N_C/P, where N_C is the number of programs proved correct (that is, those programs for which the answer is different from '*unknown*' and '*timeout*') and P is the total number of verification problems ($P = 20$, in our case), (ii) the *total time* T, that is, the time taken for proving the N_C programs correct, and (iii) the *average time*, that is, T/N_C. These experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

The data presented in Table 2 show that the delayed versions of the generalization operators we have considered have almost the same time performance and at least the same precision of their non-delayed counterparts. In particular, by using the *Gen*$_W$ operator, which is based on widening alone, our method is able to prove only 6 programs out of 20. Notably, the use of the delay mechanism in *Gen*$_{WD}$ determines a significant increase of precision with respect to *Gen*$_W$. Alternatively, precision can be increased by using the operators *Gen*$_{CHW}$ and *Gen*$_{CHWD}$, which use also convex hull. These results confirm the effectiveness of the convex hull operator which may help inferring relations among program variables, and may ease the discovery of useful program invariants, while causing (in our benchmark set) only a slight increase of the verification time.

The last column of Table 2 reports the results obtained by using BOOSTER. A distinctive feature of that tool is that it uses *loop acceleration* techniques, which allow the replacement of loops belonging to decidable classes [2] with suitable formulas. For those decidable classes BOOSTER generates proof obligations which can then be discharged by using a complete SMT solver. If the program under consideration falls outside those decidable classes, BOOSTER first runs a bounded model checking module and then, if necessary, it runs multiple parallel instances of the MCMT model checking engine [21], which also uses loop acceleration techniques together with *lazy abstraction with interpolants* for arrays. In Table 2 we have marked with (simple) the programs that BOOSTER recognizes as belonging to the decidable class of simple^0_A programs [2] for which loop acceleration performs very well.

In our experiments we found that BOOSTER is very effective at verifying the programs in our benchmark set. In some cases, it is able to prove properties of programs containing two nested loops, like the *bubblesort* program, which VeriMAP has been unable to prove using the generalization operators considered in this paper. However, in some examples the applicability and effectiveness of the loop acceleration techniques turn out to be quite sensitive to small changes in the code. For instance, we have considered 5 variants of the *init-backward-zero* program [8] (see line 15 of Table 2), and BOOSTER fails to verify 2

Example: *prog*	VeriMAP				BOOSTER
	Gen$_W$	*Gen*$_{WD}$	*Gen*$_{CHW}$	*Gen*$_{CHWD}$	
1. *bubblesort-inner*	0.67	0.85	0.70	0.88	0.01
2. *bubblesort*	*unknown*	*unknown*	*unknown*	*unknown*	0.67
3. *insertionsort-inner*	0.30	0.32	0.53	0.55	*unknown*
4. *selectionsort-inner*	*unknown*	1.34	1.16	1.38	0.15
5. *copy*	*unknown*	0.30	0.42	0.37	(simple) 0.01
6. *copy-partial*	*unknown*	0.33	0.42	0.34	(simple) 0.02
7. *copy-reverse*	*unknown*	0.36	0.68	0.63	(simple) 0.03
8. *difference*	*unknown*	0.61	1.22	1.08	(simple) 0.02
9. *sum*	*unknown*	0.65	1.30	1.13	(simple) 0.01
10. *find-first-non-null-1*	0.14	0.15	0.18	0.17	0.06
11. *find-first-non-null-2*	0.22	0.24	0.24	0.25	0.45
12. *find*	0.33	0.49	0.58	0.53	0.08
13. *init-constant*	*unknown*	0.15	0.20	0.19	(simple) 0.01
14. *init-partial-zero*	*unknown*	0.13	0.21	0.16	(simple) 0.02
15. *init-backward-zero*	*unknown*	0.11	0.24	0.21	*timeout*
16. *init-non-constant*	*unknown*	0.16	0.40	0.35	(simple) 0.02
17. *init-sequence*	*unknown*	0.63	0.93	0.85	(simple) 0.72
18. *max*	*unknown*	0.30	0.30	0.34	0.08
19. *partition*	0.49	0.53	0.56	0.55	0.12
20. *rearrange-in-situ*	*unknown*	*unknown*	0.79	0.86	0.23
precision	0.30	0.90	0.95	0.95	0.90
total time	2.15	7.65	11.06	10.82	2.04
average time	0.36	0.42	0.58	0.57	0.15

Table 2. Verification results using VeriMAP with different generalization operators and BOOSTER. Arrays have unknown dimension. Times are in seconds. (simple) denotes a program of the decidable class $\mathsf{simple}^0_{\mathcal{A}}$ [2]. The *timeout* occurs after 300 seconds. '*unknown*' denotes termination within the *timeout* without a proof or a disproof.

of these variants not falling into the class of $\mathsf{simple}^0_{\mathcal{A}}$ programs. VeriMAP can successfully verify all these variants. Similarly, a variant of the *bubblesort* program where the innermost loop moves smaller elements towards the beginning of the array (instead of moving bigger elements towards the end), could not be proved correct by BOOSTER.

We have also performed an additional experimental evaluation on the same set of problems, but using arrays of known dimension. In particular, we have considered arrays of dimension $n = 10, 25, 50$. In Table 3 we report the results obtained by running VeriMAP using the *Gen*$_{CHWD}$ generalization operator, BOOSTER, and SMACK. These experiments have been performed on an Intel Core i5-2467M 1.60GHz processor with 4GB of memory under the GNU Linux operating system. The performance of VeriMAP does not depend on the actual dimensions of the input arrays. The verification times are slighty higher than the corresponding times, shown in Table 2, obtained for programs with arrays of unknown dimension. This difference of performance is partly due to the differences of the experimental environments.

Example: *prog*	VeriMAP			BOOSTER			SMACK		
	$n=10$	$n=25$	$n=50$	$n=10$	$n=25$	$n=50$	$n=10$	$n=25$	$n=50$
1. *bubblesort-inner*	2.85	2.97	2.85	8.35	10.59	9.28	255.99	*timeout*	*timeout*
2. *bubblesort*	*timeout*	*timeout*	*timeout*	0.66	0.71	0.76	*timeout*	*timeout*	*timeout*
3. *insertionsort-inner*	1.70	1.69	1.65	*unknown*	*unknown*	*unknown*	2.71	2.60	2.93
4. *selectionsort-inner*	3.29	3.27	3.26	0.18	0.15	0.16	*timeout*	*timeout*	*timeout*
5. *copy*	1.02	1.01	1.00	0.01	0.03	0.01	18.83	*timeout*	*timeout*
6. *copy-partial*	1.01	1.07	1.06	0.03	0.02	0.03	3.87	41.83	*timeout*
7. *copy-reverse*	1.68	1.79	1.81	0.04	0.03	0.02	18.09	*timeout*	*timeout*
8. *difference*	2.38	2.42	2.46	0.03	0.03	0.02	25.15	*timeout*	*timeout*
9. *sum*	2.66	2.59	2.62	0.03	0.03	0.03	15.49	*timeout*	*timeout*
10. *find-first-non-null-1*	0.81	0.75	0.78	0.10	0.11	0.10	2.29	2.17	2.28
11. *find-first-non-null-2*	1.19	1.14	1.15	0.41	0.23	0.37	11.44	149.96	*timeout*
12. *find*	1.48	1.51	1.46	0.12	0.12	0.13	2.36	1.68	1.81
13. *init-constant*	0.66	0.59	0.61	0.02	0.01	0.03	5.42	26.07	164.80
14. *init-partial-zero*	0.58	0.55	0.58	0.02	0.02	0.02	2.99	8.99	87.17
15. *init-backward-zero*	0.59	0.58	0.59	0.14	0.31	1.62	9.37	28.39	160.87
16. *init-non-constant*	0.99	0.97	0.95	0.02	0.01	0.04	6.52	26.71	124.74
17. *init-sequence*	1.91	1.98	1.90	0.73	0.74	0.73	51.23	*timeout*	*timeout*
18. *max*	1.04	1.06	1.07	0.11	0.11	0.15	16.74	*timeout*	*timeout*
19. *partition*	2.09	2.05	2.19	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*	*timeout*
20. *rearrange-in-situ*	2.19	2.21	2.16	0.27	0.61	0.54	*timeout*	*timeout*	*timeout*
precision	0.95	0.95	0.95	0.90	0.90	0.90	0.80	0.45	0.35
total time	30.12	30.20	30.15	11.27	13.86	14.04	448.49	288.4	544.6
average time	1.59	1.59	1.59	0.63	0.77	0.78	28.03	32.04	77.80

Table 3. Verification results using VeriMAP, BOOSTER, and SMACK. Arrays have dimension $n = 10, 25, 50$. Times are in seconds. The *timeout* occurs after 300 seconds. '*unknown*' denotes termination within the *timeout* without a proof or a disproof.

The performance of BOOSTER is also generally not sensitive to variations of the array dimension, except for the *init-backward-zero* program, which it was not able to prove when using arrays of unknown dimension. We also note that for two programs the verification times are much higher than those shown in Table 2, namely *bubblesort-inner* and *partition* (which always runs out of time). This behavior is possibly due to the fact that BOOSTER, as already mentioned, makes use of a bounded model checking module before invoking MCMT. The SMACK tool, contrary to VeriMAP and BOOSTER, belongs to the family of *bounded* software verifiers. It guarantees the absence of bugs by exploring the state space up to a certain depth. SMACK first translates the LLVM intermediate representation (IR) of the program to the Boogie intermediate verification language [37], and then it uses Corral [35] as a reachability modulo theories solver. As expected, SMACK is very sensitive to the dimensions of the input arrays, except for a few problems, namely *insertionsort-inner*, *find-first-non-null-1*, and *find*, for which it does not need to reach the so called recursion bound. Moreover, even for small arrays of dimension $n = 10$, the verification times are considerably higher than those reported by the other two tools. This explains the high number of problems for which it runs out of time.

Thus, we may conclude that our transformation-based approach to the verification of programs that manipulate arrays of known or unknown dimension, is quite competitive, regarding both precision and time performance, with respect to state-of-the-art software verification methods.

7. Related Work and Conclusions

We have presented a verification method for imperative programs that manipulate integer arrays, based on an encoding of the verification task into a CLP program with constraints that represent array operations. Our method makes use of an automated strategy that guides the application of semantics preserving transformation rules, including unfolding, folding, and constraint replacement. The verification method presented in this paper is an extension of the one introduced in [12], where programs manipulate integer variables only.

The idea of encoding imperative programs into CLP programs for reasoning about their properties was presented in various papers [18, 31, 44], where it is shown that through CLP programs one can express in a simple manner both (i) the symbolic executions of the imperative programs, and (ii) the invariants that hold during these executions. The peculiarity of our work here is that we use CLP *program transformations* to prove properties, rather than symbolic execution or static analysis.

The verification method for proving properties of array manipulating programs we have presented in this paper, is related to several other methods that use abstract interpretation and theorem proving techniques.

Among the papers that use abstract interpretations for finding invariants of programs that manipulate arrays, we first mention [25]. In that paper, which builds upon [22], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is taken in [8], where a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs is introduced. In [19, 34] a predicate abstraction for inferring universally quantified properties of array elements is presented, and in [24] the authors present a similar technique that uses template-based quantified abstract domains.

The methods based on abstract interpretation construct over-approximations of the behaviour of the programs, that is, invariants implied by program executions. These methods have the advantage of being quite efficient because they fix in advance a set of assertions where the invariants are searched for, but for the same reason, they may lack flexibility as the abstraction should be re-designed when the program verification fails.

Also theorem proving techniques have been used for: (i) discovering invariants of the executions of programs that manipulate arrays, and (ii) proving the verification conditions generated from the programs to be verified. In particular, in [2, 6] satisfiability decision procedures for decidable fragments of the theory of arrays are presented. Those fragments are expressive enough to prove properties such as sortedness of arrays. In [32, 33, 40] the authors present some techniques based on theorem proving which may generate array invariants. In [49] a backward reachability analysis based on predicate abstraction and abstraction refinement is used for verifying assertions that are universally quantified over array indexes. Finally, in [2, 36] some techniques based on Satisfiability Modulo Theories (SMT) have been presented for the generation and the verification of universally quantified properties over array variables.

The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation, because no set of abstractions is fixed in advance, and the suitable assertions needed by the proof are generated on the fly, during the verification process itself. In particular, the techniques presented in [2] and related papers on decision procedures for the theory of arrays [20], have been integrated in BOOSTER [3]: a tool for verifying C-like programs handling arrays that we have used in our experimental evaluation. It exploits acceleration techniques to compute in an exact way the set of reachable states of programs with loops, provided that those programs belong to a restricted class of programs, called $\mathsf{simple}^0_{\mathcal{A}}$ programs [2]. Indeed, for programs belonging to that class, it computes in one step, the set of the reachable states for which abstraction-based approaches require several refinement steps. However, since acceleration is based on templates, it is sensitive to the syntactic presentation of the input programs, and thus the applicability of the technique may have some limitations.

Although the approach based on CLP program transformation shares many ideas and techniques with the approaches based on abstract interpretation and automated theorem proving, we believe that it has some distinctive features that can make it quite appealing. Indeed, this paper and previous works [10, 17, 44] show that one can construct a uniform framework where both the generation of verification conditions and the construction of their proofs can be viewed as instances of program transformation. The transformation-based approach is also parametric with respect to the imperative language in which the programs to be verified are written, because interpreters and proof systems can easily be written in CLP, and verification conditions can automatically be generated by program specialization (which is a particular instance of program transformation).

Moreover, optimizing transformations considered in the literature [45] can be applied to improve the efficiency of the verification task. Note also that transformations can be composed together so to derive powerful verification methods in a modular way. In particular, in [12] it is shown that the *iteration* of program specialization combined with suitable constraint propagations can significantly improve the precision of our program verification method.

Finally, we would like to mention that there are tools, such as the SMACK verifier [26], which automatically verify array manipulating programs by using bounded model checking techniques. Bounded model checkers explore the state space up-to a given bound by unrolling the control flow graph a fixed number of times only. Therefore, once provided with a suitable bound, these tools may prove the correctness of programs that manipulate arrays of *known* size. In contrast, the verification method presented in this paper and implemented in VeriMAP, as well as the techniques implemented in BOOSTER, are able to deal with arrays of *unknown* size.

As a future work we plan to extend our approach to the programs that, besides arrays, also manipulate *dynamic data structures* such as lists or heaps. This extension will be done by looking for a suitable set of constraint replacement laws that axiomatize those structures. For some specific theories we could also apply the constraint replacement rule by exploiting the results obtained by external theorem provers or SMT solvers.

Acknowledgements

We warmly thank the anonymous referees of CILC 2013 and of the Special Issue of Fundamenta Informaticae for their helpful comments and constructive criticism. This work has been partially supported by the National Group of Computing Science (GNCS-INDAM).

References

[1] 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.

[2] F. Alberti, S. Ghilardi, and N. Sharygina. Decision Procedures for Flat Array Properties. In *TACAS '14*, LNCS 8413, pages 15–30. Springer, 2014.

[3] F. Alberti, S. Ghilardi, and N. Sharygina. BOOSTER: An Acceleration-Based Verification Framework for Array Programs. In *ATVA '14*, LNCS 8837, pages 18–23. Springer, 2014.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT '12*, pages 3–11, 2012.

[5] N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS '13*, LNCS 7395, pages 105–125. Springer, 2013.

[6] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI '06*, LNCS 3855, pages 427–442. Springer, 2006.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *POPL '77*, pages 238–252. ACM, 1977.

[8] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL '11*, pages 105–118, ACM, 2011.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*, pages 84–96, ACM, 1978.

[10] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Array Programs by Transforming Verification Conditions. In *VMCAI '14*, LNCS 8318, pages 182–202. Springer, 2014.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Program verification via iterated specialization. *Science of Computer Programming*, 95, Part 2:149–175, 2014.

[13] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *ESOP'10*, LNCS 6012, pages 246–266. Springer, 2010.

[14] G. J. Duck, J. Jaffar, and N. C. H. Koh. Constraint-based program reasoning with heaps and separation. In *CP '13*, LNCS 8124, pages 282–298. Springer, 2013.

[15] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theor. Comput. Sci.*, 166:101–146, 1996.

[16] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. *Fundamenta Informaticae*, 119(3-4):281–300, 2012.

[17] 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.

[18] C. Flanagan. Automatic software model checking via constraint logic. In *Sci. Comput. Program.*, 50(1–3):253–270, 2004.

[19] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02*, pages 191–202, New York, NY, USA, 2002. ACM.

[20] 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.

[21] S. Ghilardi and S. Ranise. MCMT: A Model Checker Modulo Theories. *Proc. IJCAR '10*, LNCS 6173, pages 22–29. Springer, 2010.

[22] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL '05*, pages 338–350. ACM, 2005.

[23] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In *TACAS '12*, LNCS 7214, pages 549–551. Springer, 2012.

[24] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS '08*, LNCS 4963, pages 443–458. Springer, 2008.

[25] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI '08*, pages 339–348, 2008.

[26] A. Haran and M. Carter and M. Emmi and A. Lal and S. Qadeer and Z. Rakamarić. SMACK+Corral: A Modular Verifier. In *TACAS '15*, LNCS 9035, pages 451–454. Springer, 2015.

[27] K. S. Henriksen and J. P. Gallagher. Abstract interpretation of PIC programs through logic programming. *Proc. SCAM '06*, pages 103 – 179, 2006.

[28] G. Huet and D. C. Oppen. Equations and Rewrite Rules: A Survey. In *Formal Language Theory*: *Perspectives and Open Problems*, Academic Press, 1980.

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

[30] J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification. In *CAV '12*, LNCS 7358, pages 758–766. Springer, 2012.

[31] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP '09*, LNCS 5732, pages 454–469. Springer, 2009.

[32] R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV '07*, LNCS 4590, pages 193–206, 2007.

[33] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *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] A. Lalh, S. Qadeer and S. K. Lahiri. A Solver for Reachability Modulo Theories. In *CAV '12*, LNCS 7358, pages 427–443, 2012.

[36] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *VMCAI '13*, LNCS 7737, pages 169–188. Springer, 2013.

[37] K. Rustan M. Leino. This is Boogie 2. http://research.microsoft.com/apps/pubs/default.aspx?id=147643, 2008.

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

[39] J. McCarthy. Towards a mathematical science of computation. In C.M. Popplewell, editor, *Information Processing*: *Proceedings of IFIP 1962*, pages 21–28, Amsterdam, 1963. North Holland.

[40] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS '08*, LNCS 4963, pages 413–427, 2008.

[41] 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.

[42] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, LNCS 2304, pages 209–265. Springer, 2002.

[43] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In *LOPSTR '02*, LNCS 2664, pages 90–108. Springer, 2003.

[44] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In *SAS '98*, LNCS 1503, pages 246–261. Springer, 1998.

[45] A. Pettorossi and M. Proietti. Transformation of Logic Programs: Foundations and Techniques. In *Journal of Logic Programming*, Vol. 19,20, pages 261–320, 1994.

[46] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL '07*, LNCS 4354, pages 245–259. Springer, 2007.

[47] C. J. Reynolds. *Theories of Programming Languages*. Cambridge Univ.Press 1998.

[48] P. Rümmer, H. Hojjat, and V. Kuncak. Disjunctive interpolants for Horn-clause verification. *Proc. CAV '13*, LNCS 8044, pages 347–363. Springer, 2013.

[49] M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS '09*, LNCS 5673, pages 3–18. Springer, 2009.

[50] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.