ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

E. De Angelis, F. Fioravanti,
A. Pettorossi, M. Proietti

A RULE-BASED VERIFICATION STRATEGY FOR ARRAY MANIPULATING PROGRAMS

R. 7 2014

Emanuele De Angelis – DEC, University "G. d'Annunzio", Pescara, Italy, and Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti" del CNR, Via dei Taurini 15, I-00185 Roma, Italy. Email : deangelis@sci.unich.it.
URL: http://www.sci.unich.it/~deangelis.

Fabio Fioravanti – DEC, University "G. d'Annunzio", Pescara, Italy.
Email : fioravanti@sci.unich.it. URL: http://www.sci.unich.it/~fioravan.

Alberto Pettorossi – DICII, University of Rome Tor Vergata, Via del Politecnico 1, I-00133 Roma, Italy, and Istituto di Analisi dei Sistemi ed Informatica del CNR, Via dei Taurini 19, I-00185 Roma, Italy. Email : pettorossi@info.uniroma2.it.
URL : http://www.iasi.cnr.it/~adp.

Maurizio Proietti – Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti" del CNR, Via dei Taurini 19, I-00185 Roma, Italy. Email : maurizio.proietti@iasi.cnr.it.
URL : http://www.iasi.cnr.it/~proietti.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti",
CNR, via dei Taurini 19, 00185 ROMA, Italy

tel. ++39-0649931
fax ++39-0649937106
email: iasi@iasi.cnr.it
URL: http://www.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 $\{pre\}$ *prog* $\{post\}$ defining 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 manipulating array constraints, and also applies suitable combinations of widening and convex hull operators for generalizing 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.

Keywords: software model checking, constraint logic programming, program transformation, array programs.

1. Introduction

As it was first shown in [40], the formalism of Constraint Logic Programming (CLP) can be used for verifying properties of C-like imperative programs. By following the approach proposed in [40], the interpreter of the imperative language in which the programs are written is encoded as a CLP program, and that interpreter is specialized with respect to a given imperative program $prog$ and property φ to be verified. The result of this specialization is a new CLP program, call it *VC*, which represents the so called *verification conditions* [5] associated with the program $prog$ and the property φ. By analyzing the program *VC* (where all the references to imperative constructs have disappeared), we may infer the *loop invariants* holding during the execution of $prog$, and from these invariants we may be able to infer the property φ to be verified.

Many verification methods have been developed within the CLP paradigm. Some of those methods, following the approach proposed in [40], are based on *abstract interpretation* [7] and compute an over-approximation of the least model of the CLP program *VC* by a bottom-up evaluation of an abstraction of *VC* [1, 22, 37]. Other methods use goal directed evaluation of CLP programs combined with other techniques such as the *interpolation* technique [15, 18, 27, 28]. Some other methods, like those presented in [4, 23, 42, 44], use CLP (also called *constrained Horn clauses* in those papers) together with various reasoning techniques such as the *CounterExample-Guided Abstraction Refinement* (CEGAR) and the *Satisfiability Modulo Theory* (SMT) that have been developed in the areas of Software Model Checking and Automated Theorem Proving.

In this paper we follow the verification approach based on transformations of CLP programs which has been presented in [10, 12]. First, we consider the partial correctness property of the given imperative program *prog*, defined as the Hoare triple $\{pre\}\ prog\ \{post\}$, and we encode its negation as a predicate `incorrect` defined by a CLP program P. Then, similarly to [40], we generate a CLP program *VC* representing the verification conditions for *prog*, by specializing P with respect to the CLP representation of *prog*. At this point our verification method departs from the method presented in [40] and all other verification methods considered above. In particular, we proceed by applying a sequence of equivalence preserving transformations to *VC* with the objective of deriving a CLP program Tsuch 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* we derive a CLP program containing one or more clauses of the form `incorrect :- G`, where `G` is a non-empty conjunction, and we are able to conclude neither that $prog$ is correct nor that $prog$ is incorrect.

The verification method we propose provides a *uniform* framework where one can both construct and manipulate the verification conditions that are required to prove the correctness of the given programs. Moreover, that framework is *parametric* with respect to the syntax and the semantics of the programs to be verified, and optimizing transformations considered in the literature [41] can be applied to improve the efficiency of the verification method. Finally, transformations can easily be composed together into a sequence of transformations, so as to derive very sophisticated verification methods. For instance, in [10] it is shown that the *iteration* of program specialization can significantly improve the precision of our program verification method.

The main contributions of this paper are the following ones.

(1) We provide a method that given any C-like imperative program that manipulates integers and in-

teger arrays, generates the verification conditions where the read and write operations on arrays are represented as constraints. Then, we show how the verification conditions can be suitably manipulated using *constraint replacement rules* together with the familiar *unfold/fold transformation rules* for CLP programs [16], with the objective of proving properties of the given imperative program.
(2) We propose a powerful *transformation strategy* for guiding the application of the constraint replacement and unfold/fold transformation rules. In particular, we design a novel *generalization strategy for array constraints* that automatically introduces, during CLP transformation, 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 also makes use the widening operator and the convex hull operator for linear constraints that have been introduced in the field of *abstract interpretation* [9].
(3) Finally, we present an *implementation* of our verification method in the VeriMAP system [13], and we demonstrate its good performance 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 in the domain of the integers and integer arrays. Then, in Section 3 we introduce some transformation rules for manipulating CLP(Array) programs. Besides the usual *unfolding* and *folding* rules, we consider the *constraint replacement* rule, which allows us to replace constraints by equivalent ones in the theory of arrays [6, 20, 35]. 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 designed for applying the transformation rules with the objective of obtaining a proof (or a disproof) of the properties of interest. The transformation strategy may introduce some auxiliary predicates by using a *generalization strategy* that extends to CLP(Array) the generalization strategies for CLP programs over integers or reals [17]. Finally, in Section 6 we present the experimental results obtained by using the VeriMAP system [13].

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

In this section we recall some basic notions concerning Constraint Logic Programming (CLP), and we introduce a set, called CLP(Array), of CLP programs with constraints on integers and integer arrays. More notions concerning CLP can be found in [26]. For reasons of simplicity we will consider one-dimensional arrays only.

If $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer variables and coefficients, then $\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, and $\mathtt{p}_1 > \mathtt{p}_2$ are *atomic integer constraints* (sum and multiplication are denoted by `+` and `*`, respectively). The dimension `n` of an array `a` is represented by the predicate $\mathtt{dim}(\mathtt{a}, \mathtt{n})$. The read and write operations on arrays are represented by the predicates `read` and `write`, respectively. We have that $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$ denotes that the `i`-th element of the array `a` is the value `v`, and $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$ denotes that the array `b` is equal to the array `a`, except that its `i`-th element is the value `v`. For reasons of simplicity, we assume that indexes of arrays and values of array elements are integers, but our program verification method is parametric with respect to the index and value domains. Note that the success or the failure of the program verification task for a given array program may depend on the constraint domain and also on the constraint solver one uses.

An *atomic array constraint* is an atom of the following form: either $\mathtt{dim}(\mathtt{a}, \mathtt{n})$, or $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$, or $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$. A *constraint* is either `true`, or an atomic (integer or array) constraint, or a *conjunction* of constraints. An *atom* is an atomic formula of the form $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_m)$, where `p` is a predicate symbol

not in $\{=, \geq, >, \mathtt{dim}, \mathtt{read}, \mathtt{write}\}$ and $\mathtt{t}_1, \ldots, \mathtt{t}_\mathtt{m}$ 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` is called the *head*, and `c, B` is called the *body*. We assume that in every clause all integer arguments in its head are distinct variables. A clause `A :- c` is called a *constrained fact*. If `c` is `true`, then it is omitted and the constrained fact is called a *fact*. A *goal* is a formula of the form `:- c, B` (standing for $c \wedge B \rightarrow \mathit{false}$ or, equivalently, $\neg(c \wedge B)$). 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.

Now we define the semantics of CLP(Array) programs. An *$\mathcal{A}$-interpretation* I is a set D, together with a function f in $D^n \rightarrow D$ for each function symbol `f` of arity n, and a relation p on D^n for each predicate symbol `p` of arity n, such that:

(i) the set D is the Herbrand universe [33] constructed out of the set $\mathbb{Z}$ of the integers, the constants, and the function symbols different from + and *,

(ii) I assigns to the symbols $+, *, =, \geq, >$ the usual meaning in $\mathbb{Z}$,

(iii) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$, for all integers `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 [33] 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 $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. We assume that we are given a solver for the theory of arrays (for instance, a solver implementing the algorithms described in [6, 20]) to check the satisfiability and the entailment of constraints in $\mathcal{A}$. By $\mathit{vars}(\varphi)$ we denote the set of all free variables of the formula φ.

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 transformations that, under suitable conditions, 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, [16]).

Let P be any given CLP(Array) program.

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

Unfolding Rule. Let us consider a clause C of the form $\mathtt{H\ :-\ c\,,L\,,A\,,R}$, where $\mathtt{H}$ and $\mathtt{A}$ are atoms, $\mathtt{c}$ is a constraint, and $\mathtt{L}$ and $\mathtt{R}$ are (possibly empty) conjunctions of atoms. Let us also consider the set $\{\mathtt{K}_i\ \mathtt{:-}\ \mathtt{c}_i\,,\mathtt{B}_i \mid i=1,\ldots,m\}$ of the (renamed apart) clauses of P such that, for $i=1,\ldots,m$, $\mathtt{A}$ 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 $\{(\mathtt{H\ :-\ c}\,,\mathtt{c}_i\,,\mathtt{L}\,,\mathtt{B}_i\,,\mathtt{R})\,\vartheta_i \mid i=1,\ldots,m\}$ of clauses.

Constraint Replacement Rule. Let us consider a clause C of the form: $\mathtt{H\ :-\ c_0}$, $\mathtt{B}$, 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 $\mathtt{c}_1,\ldots,\mathtt{c}_\mathtt{n}$, respectively.

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

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

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

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

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 $\mathtt{dim}$.

Folding Rule. Given a clause E: $\mathtt{H\ :-\ e\,,\ L\,,\ A\,,\ R}$ and a clause D: $\mathtt{K\ :-\ d\,,\ D}$ 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 $\mathtt{H\ :-\ e\,,\ L\,,\ K}\,\vartheta\mathtt{\,,\ R}$.

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 correctness results for the unfold/fold transformation rules of CLP programs [16].

Theorem 3.1. (*Correctness 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 [16]. Informally, that assumption forbids the replacement of a definition clause of the form $\mathtt{A\ :-\ B}$ by the clause $\mathtt{A\ :-\ A}$ that is obtained by folding clause $\mathtt{A\ :-\ B}$ using $\mathtt{A\ :-\ B}$ 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 our language is defined by 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 $\mathtt{a}_0 \ldots \mathtt{a}_{\mathtt{n}-1}$ of integers, where n is the dimension of the array a. The definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [43] 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 formalize the *program incorrectness problem*, defined by the Hoare triple $\{\varphi_{init}\}\ prog\ \{\neg\varphi_{error}\}$, that is, the problem of checking whether or not an execution of the program *prog* may lead 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*), as the problem of checking whether or not the nullary predicate incorrect (standing for *false*) is a consequence of the CLP(Array) program P defined by 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 initConf(X), errorConf(X), and tr(X, Y) encoding, respectively, an initial configuration, an error configuration, and the transition relation $\Longrightarrow$ between configurations. The predicate reach(Y) holds if a configuration Y can be reached from an initial configuration.

As an example of 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 $\langle\!\langle \ell\colon c, \delta \rangle\!\rangle$, 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,array(A),FA), write(FA,I,V,FA1),
    update(D,array(A),FA1,D1), nextlab(L,L1), at(L1,C).
```

In the body of this clause: (i) eval(IE, D, I) returns the value I of the index IE in the environment D, (ii) eval(E,D,V) returns the value V of the expression E in the environment D, (iii) lookup(D,array(A),FA) returns the value FA of the array A in the environment D, (iv) update(D, array(A), FA1, D1) returns the new value D1 of the environment D after the array assignment, (v) nextlab(L, L1) returns the label L1 of the command following the command with label L in the encoding of the given program *prog*, and (vi) at(L1, C) returns the command C with label L1 in that encoding.

As shown in [10], similar clauses for the predicate tr can be defined for the other kinds of commands, such as conditionals and jumps, which may occur in imperative programs.

The imperative program *prog* is correct with respect to φ_{init} and φ_{error} iff incorrect $\notin M(P)$ (or, equivalently, $P \not\models$ incorrect), where $M(P)$ is the least $\mathcal{A}$-model of program P (see Section 2).

Our verification method performs a sequence of applications of the unfold/fold rules presented in Section 3 starting from program P and, as a consequence of Theorem 3.1, we have that, for each program P' obtained from P by a sequence of applications of the transformation rules, $\texttt{incorrect} \in M(P)$ iff $\texttt{incorrect} \in M(P')$.

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:
(i) the *Generation of Verification Conditions* (*VCGen*), and
(ii) the *Transformation of Verification Conditions* (*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 VC, representing the *verification conditions* for *prog*. For program VC we have that: (i) $\texttt{incorrect} \in M(P)$ iff $\texttt{incorrect} \in M(VC)$, and (ii) `tr` does not occur in VC. The specialization of P is obtained by applying a variant of the strategy presented in [10] for the operation called *the removal of the interpreter*. The main difference with respect to [10] is that the CLP programs considered in this paper 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. All occurrences of the `dim` predicate are eliminated by replacing them by suitable integer constraints on indexes. The verification conditions VC are said to be *satisfiable* iff $\texttt{incorrect} \notin M(VC)$ (or equivalently $VC \not\models \texttt{incorrect}$). Thus, the satisfiability of the verification conditions for *prog* guarantees that *prog* is correct with respect to φ_{init} and φ_{error}.

During *VCTransf*, through further transformations, we check the satisfiability of the verification conditions generated at the end of *VCGen*. In Section 5 we will describe this step in detail and we will present the strategy that guides the transformation rules during this step.

Now let us see our verification method in action in an example. 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: $\ell_0\colon\ i = 1;$
$\ell_1\colon\ \texttt{while}\ (i{<}n)\ \{\ a[i] = a[i{-}1] + 1;\ i = i + 1;\ \};$
$\ell_h\colon\ \texttt{halt}$

We consider the Hoare triple $\{\varphi_{init}(i,n,a)\}\ \mathit{SeqInit}\ \{\neg\varphi_{error}(n,a)\},$ where:
(i) $\varphi_{init}(i,n,a)$ is $i{\geq}0\ \wedge\ n{=}\mathit{dim}(a)\ \wedge\ n{\geq}1$, 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. $\texttt{phiInit(I,N,A)}$ `:-` $\texttt{I}{\geq}\texttt{0},\ \texttt{dim(A,N)},\ \texttt{N}{\geq}\texttt{1}.$
2. $\texttt{phiError(N,A)}$ `:-` $\texttt{K}{=}\texttt{J}{+}\texttt{1},\ \texttt{J}{\geq}\texttt{0},\ \texttt{J}{\leq}\texttt{N}{-}\texttt{2},\ \texttt{U}{\geq}\texttt{V},\ \texttt{read(A,J,U)},\ \texttt{read(A,K,V)}.$

The clauses defining the predicates `initConf` and `errorConf` which specify the initial and the error configurations, respectively, are as follows (where `D` stands for a suitable tuple of terms):

3. $\texttt{initConf(cf(cmd(l}_0\texttt{,Cmd),D))}$ `:-` $\texttt{at(l}_0\texttt{,Cmd)},\ \texttt{environment(D)},\ \texttt{phiInit(D)}.$
4. $\texttt{errorConf(cf(cmd(l}_h\texttt{,Cmd),D))}$ `:-` $\texttt{at(l}_h\texttt{,Cmd)},\ \texttt{environment(D)},\ \texttt{phiError(D)}.$

The predicates `at` and `environment` are defined by the facts:

$\texttt{at(l}_0\texttt{,asgn(int(i),int(1)))}.$
$\texttt{at(l}_h\texttt{,halt)}.$
$\texttt{environment([[int(i),I],[int(n),N],[array(a),A]])}.$

Now we apply the *VCGen* step of our verification method. After the removal of the interpreter from program P we obtain the following program VC:

5. $\texttt{incorrect} \texttt{ :- } \texttt{K}=\texttt{J}+1,\ \texttt{J}\geq 0,\ \texttt{J}\leq\texttt{N}-2,\ \texttt{U}\geq\texttt{V},\ \texttt{N}\leq\texttt{I},\ \texttt{read}(\texttt{A},\texttt{J},\texttt{U}),\ \texttt{read}(\texttt{A},\texttt{K},\texttt{V}),\ \texttt{p}(\texttt{I},\texttt{N},\texttt{A})\texttt{.}$
6. $\texttt{p}(\texttt{I},\texttt{N},\texttt{A}) \texttt{ :- } 1\leq\texttt{H},\ \texttt{H}\leq\texttt{N}-1,\texttt{G}=\texttt{H}-1,\ \texttt{I}=\texttt{H}+1,\ \texttt{Z}=\texttt{W}+1,\ \texttt{read}(\texttt{B},\texttt{G},\texttt{W}),\texttt{write}(\texttt{B},\texttt{H},\texttt{Z},\texttt{A}),\texttt{p}(\texttt{H},\texttt{N},\texttt{B})\texttt{.}$
7. $\texttt{p}(\texttt{I},\texttt{N},\texttt{A}) \texttt{ :- } \texttt{I}=1,\ \texttt{N}\geq 1\texttt{.}$

The CLP(Array) program VC expresses the verification conditions for *SeqInit* and the formula defining the predicate p is an invariant of the while-loop.

In the following section we will present the *VCTransf* transformation step of our verification method. Note that in order to prove program correctness, this step cannot be avoided in favor of a standard evaluation method, because the least $\mathcal{A}$-model $M(VC)$ may be infinite and both the bottom-up and top-down evaluation of the goal `:- incorrect` may not terminate (indeed, this is the case in our example above).

5. A Strategy for Transforming the Verification Conditions

The verification conditions expressed as the CLP(Array) program VC generated by the *VCGen* step are satisfiable iff $\texttt{incorrect} \notin M(VC)$. Our verification method is based on the fact that by transforming the program VC using rules that preserve the least $\mathcal{A}$-model, we get a new CLP(Array) program, call it T, that expresses equisatisfiable verification conditions.

The *VCTransf* step has the objective of showing, through further transformations, that *either* the verification conditions generated by *VCGen* are satisfiable (that is, $\texttt{incorrect} \notin M(VC)$, and hence *prog* is correct with respect to φ_{init} and φ_{error}), *or* they are unsatisfiable (that is, $\texttt{incorrect} \in M(VC)$, and hence *prog* is not correct with respect to φ_{init} and φ_{error}). To this aim, *VCTransf* propagates the constraints occurring in φ_{init} and/or φ_{error}, so as to derive from program VC a program T where the predicate `incorrect` is defined by either (i) the fact '`incorrect`' (in which case the verification conditions are unsatisfiable and *prog* is incorrect), or (ii) the empty set of clauses (in which case the verification conditions are satisfiable 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], in this case we can iterate *VCTransf*, alternating the propagation of the constraints in φ_{init} and/or φ_{error}, 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 *Transform* strategy shown in Figure 1. *Transform* 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 *Transform* with the *Reversal* transformation described in [10]. For lack of space we do not present this extra transformation here.

The program VC which is the input of the *Transform* strategy, is a *linear* CLP(Array) program (we can show, in fact, that *VCGen* always generates a linear program).

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

Input: A linear CLP(Array) program VC.
Output: Program T such that $\texttt{incorrect} \in M(VC)$ iff $\texttt{incorrect} \in M(T)$.

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 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 contains 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 $\texttt{q}(\ldots)$ `:- c` where `q` is either `p` or a predicate on which `p` depends.

Figure 1. The *Transform* 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* example of Section 4, and let VC be the CLP program made out of clauses 5, 6, and 7. The *Transform* strategy starts off by unfolding clause 5 w.r.t. the atom $\texttt{p}(\texttt{I}, \texttt{N}, \texttt{A})$. We get the clause:

8. $\texttt{incorrect :- K}=\texttt{J}+1,\ \texttt{J} \geq 0,\ \texttt{J} \leq \texttt{N}-2,\ \texttt{U} \geq \texttt{V},\ \texttt{N} \leq \texttt{I},\ 1 \leq \texttt{H},\ \texttt{H} \leq \texttt{N}-1, \texttt{G}=\texttt{H}-1, \texttt{I}=\texttt{H}+1, \texttt{Z}=\texttt{W}+1,$
$\texttt{read}(\texttt{A}, \texttt{J}, \texttt{U}),\ \texttt{read}(\texttt{A}, \texttt{K}, \texttt{V}),\ \texttt{read}(\texttt{B}, \texttt{G}, \texttt{W}),\ \texttt{write}(\texttt{B}, \texttt{H}, \texttt{Z}, \texttt{A}),\ \texttt{p}(\texttt{H}, \texttt{N}, \texttt{B}).$

where `B` denotes the array from which the output array `A` computed by the *SeqInit* program is obtained. 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 constraints occurring in clauses 5 and 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 remove `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, say $\mathtt{H}$:- $\mathtt{c}, \mathtt{G}$, 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. Note that this replacement process which is, in general, nondeterministic, always terminates (see Theorem 5.1).

(RR1) If $\mathtt{c} \sqsubseteq (\mathtt{I}=\mathtt{J})$ then
replace: $\mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$ by: $\mathtt{U}=\mathtt{V},\ \mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U})$

(RR2) If $\mathtt{c} \equiv (\mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{U}), \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}), \mathtt{d})$, $\mathtt{d} \not\sqsubseteq (\mathtt{I}\neq\mathtt{J})$, and $\mathtt{d} \sqsubseteq (\mathtt{U}\neq\mathtt{V})$ then
add to $\mathtt{c}$ the constraint: $\mathtt{I}\neq\mathtt{J}$

(WR1) If $\mathtt{c} \sqsubseteq (\mathtt{I}=\mathtt{J})$ then
replace: $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$ by: $\mathtt{U}=\mathtt{V},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B})$

(WR2) If $\mathtt{c} \sqsubseteq (\mathtt{I}\neq\mathtt{J})$ then
replace: $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$ by: $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$

(WR3) If $\mathtt{c} \not\sqsubseteq \mathtt{I}=\mathtt{J}$ and $\mathtt{c} \not\sqsubseteq \mathtt{I}\neq\mathtt{J}$ then
replace: $\mathtt{H}$:- $\mathtt{c},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}),\ \mathtt{G}$
by: (α) $\mathtt{H}$:- $\mathtt{c},\ \mathtt{I}=\mathtt{J},\ \mathtt{U}=\mathtt{V},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{G}$
and (β) $\mathtt{H}$:- $\mathtt{c},\ \mathtt{I}\neq\mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V}),\ \mathtt{G}$

Rules RR1 and RR2 are derived from the array axiom A1 (see Section 3), and rules WR1–WR3 are derived from the array axioms A2 and A3 (see Section 3).

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

Then, since neither $\mathtt{K}=\mathtt{H}$ nor $\mathtt{K}\neq\mathtt{H}$ 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 $\mathtt{incorrect}$:- $\mathtt{K}=\mathtt{J}+1,\ \mathtt{J}\geq 0,\ \mathtt{K}\leq\mathtt{H},\ \mathtt{G}=\mathtt{H}-1,\ \mathtt{N}=\mathtt{H}+1,\ \mathtt{Z}=\mathtt{W}+1,\ \mathtt{U}\geq\mathtt{V},$
$\underline{\mathtt{K}=\mathtt{H}}, \underline{\mathtt{Z}=\mathtt{V}},\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{U}),\ \mathtt{read}(\mathtt{B},\mathtt{G},\mathtt{W}),\ \underline{\mathtt{write}(\mathtt{B},\mathtt{H},\mathtt{Z},\mathtt{A})},\ \mathtt{p}(\mathtt{H},\mathtt{N},\mathtt{B}).$

8r.3 $\mathtt{incorrect}$:- $\mathtt{K}=\mathtt{J}+1,\ \mathtt{J}\geq 0,\ \mathtt{K}\leq\mathtt{H},\ \mathtt{G}=\mathtt{H}-1,\ \mathtt{N}=\mathtt{H}+1,\ \mathtt{Z}=\mathtt{W}+1,\ \mathtt{U}\geq\mathtt{V},$
$\underline{\mathtt{K}\neq\mathtt{H}},\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{U}),\ \underline{\mathtt{read}(\mathtt{B},\mathtt{K},\mathtt{V})},\ \mathtt{read}(\mathtt{B},\mathtt{G},\mathtt{W}),\ \underline{\mathtt{write}(\mathtt{B},\mathtt{H},\mathtt{Z},\mathtt{A})},\ \mathtt{p}(\mathtt{H},\mathtt{N},\mathtt{B}).$

Finally, since $\mathtt{G}=\mathtt{J}$ is entailed by the constraint in clause 8r.2, we apply rule RR1 to clause 8r.2 and we add the constraint $\mathtt{W}=\mathtt{U}$ to its body, hence deriving the unsatisfiable constraint '$\mathtt{W}=\mathtt{U}, \mathtt{Z}=\mathtt{W}+1, \mathtt{Z}=\mathtt{V}, \mathtt{U}\geq\mathtt{V}$'. Thus, clause 8r.2 is removed by the subsequent CLAUSE REMOVAL phase. From clause 8r.3, by rewriting '$\mathtt{K}\leq\mathtt{H},\ \mathtt{K}\neq\mathtt{H}$' as '$\mathtt{K}\leq\mathtt{H}-1$', we get:

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

We have the following result concerning the CONSTRAINT REPLACEMENT phase.

Theorem 5.1. Let U be a clause obtained after the UNFOLDING phase. Then, the execution of the CONSTRAINT REPLACEMENT phase on U terminates.

Proof:
Let us define a relation, denoted $\prec$, on the variables of a clause as follows: $\texttt{A}\prec\texttt{B}$ iff the constraint $\texttt{write(A, I, U, B)}$ occurs in the clause. It can be shown that in every clause derived by the UNFOLDING phase during the application of the *Transform* strategy, the transitive closure $\prec^+$ of $\prec$ is irreflexive (because the result of a write operation on any array A is denoted by a fresh new array variable, B).

The termination of the CONSTRAINT REPLACEMENT phase is a consequence of the following facts:
(i) in rules RR1 and WR1 and in clause (α) of rule WR3, the number of read constraints strictly decreases and no other rule increases that number,
(ii) no rule introduces new variables (thus, neither rule RR2 nor clause (β) of rule WR3 can introduce an unbounded number of new integer constraints of the form $\texttt{I}\neq\texttt{J}$), and
(iii) in rule WR2 the constraint $\texttt{read(B, J, V)}$ is replaced by the constraint $\texttt{read(A, J, V)}$ with $\texttt{A}\prec\texttt{B}$. □

5.3. Definition and Folding

The DEFINITION & FOLDING phase introduces new predicate definitions by suitable generalizations of the constraints. Generalization has to cope with two somewhat conflicting issues: (1) the termination of the *Transform* strategy, and (2) the computation of a maximally precise representation of the configurations that are reachable in a backward way from the error configuration.

The termination of *Transform* is always guaranteed by using the generalization technique presented below, which enforces 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. However, maximal precision cannot be guaranteed, because we may introduce an overly general new predicate definition whose unfolding may generate constrained facts, even if the given array manipulating program is correct. Clearly, due to the undecidability of program correctness, no generalization technique can guarantee termination and maximal precision at the same time.

The DEFINITION & FOLDING phase works as follows. Let $C1$ in $R(C)$ be a clause of the form $\texttt{H :- c, p(X)}$. We assume that *Defs* is structured 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: $\texttt{newp(X) :- d, p(X)}$ such that $\mathit{vars}(\texttt{d})\subseteq\mathit{vars}(\texttt{c})$ and $\texttt{c}\sqsubseteq\texttt{d}$, then we fold $C1$ using D. Otherwise, we introduce a clause of the form $\texttt{newp(X) :- gen, p(X)}$ where: (i) newp is a predicate symbol occurring neither in the initial program nor in *Defs*, and (ii) gen is a constraint such that $\mathit{vars}(\texttt{gen})\subseteq\mathit{vars}(\texttt{c})$ and $\texttt{c}\sqsubseteq\texttt{gen}$. The constraint gen is called a *generalization* of the constraint c.

Many different generalizations of constraints exist. Now we propose an algorithm for computing one such generalization, and in Section 6 we show that this algorithm is effective in several program verification tasks taken from the literature.

The Generalization Algorithm

Input: (i) A clause C, (ii) a clause in $R(C)$ of the form $\mathtt{H}$ `:-` $\mathtt{c}$`,` $\mathtt{p(X)}$, and (iii) a tree *Defs* of predicate definitions.

Output: A constraint $\mathtt{gen}$ which is a *generalization* of the constraint $\mathtt{c}$.

Let $\mathtt{c}$ 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 $\mathtt{c}$ are distinct variables not occurring in $\mathtt{X}$ (this condition can always be fulfilled by adding suitable 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 $\mathtt{Y}$ 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 $\mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$ constraints such that either (i) $\mathtt{A}$ does not occur in $\mathtt{X}$, or (ii) $\mathtt{V}$ 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 $\mathtt{H}_0$ `:-` $\mathtt{i}_0, \mathtt{r}_0, \mathtt{p(X)}$ such that $\mathtt{r}_0, \mathtt{p(X)}$ is a subconjunction of $\mathtt{r}_2, \mathtt{p(X)}$,
   then compute a generalization $\mathtt{g}$ of the constraints $\mathtt{i}_2$ and $\mathtt{i}_0$ such that $\mathtt{i}_2 \sqsubseteq \mathtt{g}$, by using a *generalization operator* for linear constraints (we refer to [9, 17, 39] for generalization operators based on *widening*, *convex hull*, and *well-quasi orderings*). Define the constraint $\mathtt{gen}$ as $\mathtt{g}, \mathtt{r}_0$;
   else define the constraint $\mathtt{gen}$ as $\mathtt{i}_2, \mathtt{r}_2$.

Now, let us make some remarks on this Generalization Algorithm, and show its correctness.

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 $\mathtt{i}_1 \sqsubseteq \mathtt{i}_2$ 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 $\mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$, where $\mathtt{A}$ does not appear in $\mathtt{X}$, is motivated by the fact that $\mathtt{A}$ can be treated as an existentially quantified variable, and $\exists \mathtt{A}.\, \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$ holds for all $\mathtt{I}$ and $\mathtt{V}$. The deletion of constraints of the form $\mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$, where $\mathtt{V}$ does not occur in $\mathtt{i}_2$, is motivated by the fact that $\mathtt{V}$ can be treated as an existentially quantified variable and, since by construction $\mathtt{i}_2$ enforces that the index $\mathtt{I}$ is within bounds, we have that $\exists \mathtt{V}.\, \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$ holds for all $\mathtt{A}$ and $\mathtt{I}$. 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 $\mathtt{g}$ 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 *Transform* 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 $\mathtt{gen}$ 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 $\mathtt{c} \sqsubseteq \mathtt{gen}$.

Let us continue our program verification example by performing, starting from clause 9, the DEFINITION & FOLDING phase of the *Transform* 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. $\mathtt{incorrect} \,\mathtt{:-}\, \mathtt{K}{=}\mathtt{J}{+}1,\ \mathtt{J}{\geq}0,\ \mathtt{K}{\leq}\mathtt{I}{-}1,\ \mathtt{G}{=}\mathtt{I}{-}1,\ \mathtt{N}{=}\mathtt{I}{+}1,\ \mathtt{Z}{=}\mathtt{W}{+}1,\ \mathtt{U}{\geq}\mathtt{V},$
$\mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{K},\mathtt{V}),\ \mathtt{read}(\mathtt{A},\mathtt{G},\mathtt{W}),\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{Z},\mathtt{A1}),\ \mathtt{p}(\mathtt{I},\mathtt{N},\mathtt{A}).$

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

$\mathtt{K}{=}\mathtt{J}{+}1,\ \mathtt{J}{\geq}0,\ \mathtt{K}{\leq}\mathtt{I}{-}1,\ \mathtt{N}{=}\mathtt{I}{+}1,\ \mathtt{U}{\geq}\mathtt{V},\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{K},\mathtt{V}).$

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

10. $\mathtt{new1}(\mathtt{I},\mathtt{N},\mathtt{A}) \,\mathtt{:-}\, \mathtt{K}{=}\mathtt{J}{+}1,\ \mathtt{J}{\geq}0,\ \mathtt{J}{\leq}\mathtt{N}{-}2,\ \mathtt{J}{\leq}\mathtt{I}{-}2,\ \mathtt{N}{\leq}\mathtt{I}{+}1,\ \mathtt{U}{\geq}\mathtt{V},$
$\mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{K},\mathtt{V}),\ \mathtt{p}(\mathtt{I},\mathtt{N},\mathtt{A}).$

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

11. $\mathtt{incorrect} \,\mathtt{:-}\, \mathtt{K}{=}\mathtt{J}{+}1,\ \mathtt{J}{\geq}0,\ \mathtt{K}{\leq}\mathtt{I}{-}1,\ \mathtt{G}{=}\mathtt{I}{-}1,\ \mathtt{N}{=}\mathtt{I}{+}1,\ \mathtt{Z}{=}\mathtt{W}{+}1,\ \mathtt{U}{\geq}\mathtt{V},$
$\mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{U}),\ \mathtt{read}(\mathtt{A},\mathtt{K},\mathtt{V}),\ \mathtt{read}(\mathtt{A},\mathtt{G},\mathtt{W}),\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{Z},\mathtt{A1}),\ \mathtt{new1}(\mathtt{I},\mathtt{N},\mathtt{A}).$

5.4. Termination and Soundness of the Transformation Strategy

We have the following results concerning the *Transform* strategy.

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

Proof:
(Sketch) The termination of the *Transform* strategy is based on the following facts.
(i) The UNFOLDING, CONSTRAINT REPLACEMENT, CLAUSE REMOVAL, and DEFINITION & FOLDING phases terminate. In particular, constraint satisfiability and entailment are decidable and can be checked by a terminating solver (note that completeness of the solver is not necessary for the termination of the *Transform* strategy), and each sequence of constraint replacements terminates (see Section 5.2).
(ii) The while-loop of the *Transform* strategy terminates because the set of new predicate definitions that, during the execution of the strategy, is introduced by executions of DEFINITION & FOLDING phase is finite. Indeed, by construction, each predicate definition is of the form $\mathtt{H} \,\mathtt{:-}\, \mathtt{i},\mathtt{r},\mathtt{p}(\mathtt{X})$, where:

- `X` is a tuple of variables of fixed length,
- `i` is an integer constraint,
- `r` is a conjunction of array constraints of the form $\mathtt{read}(\mathtt{A},\mathtt{I},\mathtt{V})$, where `A` is a variable in `X` and the variables `I` and `V` occur in `i` only (see Step 1 of the Generalization Algorithm),

- the cardinality of r is bounded, because the Generalization Algorithm does *not* introduce a clause $\mathtt{newp(X)}$:- $\mathtt{i_3, r_3, p(X)}$ if there exists an ancestor clause of the form $\mathtt{H_0}$:- $\mathtt{i_0, r_0, p(X)}$ such that $\mathtt{r_0, p(X)}$ is a subconjunction of $\mathtt{r_2, p(X)}$ (see Step 4 of the Generalization Algorithm),
- we assume that the generalization operator on integer constraints has the following *finiteness* property: only finite chains of generalizations of any given integer constraint can be generated by applying the operator. The already mentioned generalization operators presented in [9, 17, 39] satisfy this finiteness property.

The soundness of the strategy with respect to the least $\mathcal{A}$-model semantics follows from Theorem 3.1 of Section 3. Note that, in particular, every clause defining a new predicate introduced during the DEFINITION & FOLDING phase is unfolded once during the execution of the strategy. □

Let us now conclude the verification of the *SeqInit* program. The *Transform* 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. $\mathtt{new1(I,N,A)}$:- $\mathtt{K{=}J{+}1,\ J{\geq}0,\ J{\leq}N{-}2,\ J{\leq}I{-}2,\ N{\leq}I{+}1,\ U{\geq}V,}$
$\mathtt{1{\leq}H,\ H{\leq}N{-}1,\ G{=}H{-}1,\ I{=}H{+}1,\ Z{=}W{+}1,}$
$\mathtt{read(A,J,U),\ read(A,K,V),\ read(B,G,W),\ write(B,H,Z,A),\ p(H,N,B).}$

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

13. $\mathtt{new1(I,N,A)}$:- $\mathtt{K{=}J{+}1,\ I{=}H{+}1,\ Z{=}W{+}1,\ G{=}H{-}1,}$
$\mathtt{N{\leq}H{+}2,\ K{\leq}H{-}1,\ K{\geq}1,\ N{\geq}H{+}1,\ U{\geq}V,}$
$\mathtt{read(B,J,U),\ read(B,K,V),\ read(B,G,W),\ write(B,H,Z,A),\ p(H,N,B).}$

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. $\mathtt{new2(I,N,A)}$:- $\mathtt{K{=}J{+}1,\ J{\geq}0,\ J{\leq}I{-}2,\ N{\geq}K{-}1,\ U{\geq}V,}$
$\mathtt{read(A,J,U),\ read(A,K,V),\ p(I,N,A).}$

By folding clause 13 using clause 14, we get:

15. $\mathtt{new1(I,N,A)}$:- $\mathtt{K{=}J{+}1,\ I{=}H{+}1,\ Z{=}W{+}1,\ G{=}H{-}1,\ N{\leq}H{+}2,}$
$\mathtt{K{\leq}H{-}1,\ K{\geq}1,\ N{\geq}H{+}1,\ U{\geq}V,}$
$\mathtt{read(B,J,U),\ read(B,K,V),\ read(B,G,W),\ write(B,H,Z,A),\ new2(H,N,B).}$

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. $\mathtt{new2(I,N,A)}$:- $\mathtt{K{=}J{+}1,\ I{=}H{+}1,\ Z{=}W{+}1,\ G{=}H{-}1,}$
$\mathtt{J{\leq}H{-}1,\ K{\geq}1,\ N{\geq}H{+}1,\ U{\geq}V,}$
$\mathtt{read(B,J,U),\ read(B,K,V),\ read(B,G,W),\ write(B,H,Z,A),\ new2(H,N,B).}$

The final 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 pro-

gram T, and we conclude that the program $SeqInit$ is correct with respect to the given φ_{init} and φ_{error} properties.

6. Experimental Evaluation

We have implemented the transformation strategy described in Section 5 as a module of the VeriMAP software model checker [13] and we have performed an experimental evaluation of our method on a benchmark set of programs acting on arrays, mostly taken from the literature [5, 14, 25, 32]. The results of our experiments, which are summarized in Tables 1 and 2, show that our approach is quite effective and efficient in practice.

The VeriMAP tool consists of: (i) a module, based on the C Intermediate Language (CIL) [38], which translates a C program together with the initial and error configurations, into a set of CLP(Array) facts, and (ii) a module for CLP(Array) program transformation that generates the verification conditions and applies the *Transform* strategy. The latter module is implemented using the MAP system [34], a tool for transforming constraint logic programs written in SICStus Prolog.

We now briefly discuss the programs we have used for our experimental evaluation (see Table 1 where in the last column we have indicated the properties we have verified).

Programs *bubblesort-inner*, *insertionsort-inner*, and *selectionsort-inner* are based on textbook implementations of sorting algorithms. Programs *copy* and *copy-partial* perform the element-wise copy of the entire input array or a portion of it, respectively. The 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* and *first-non-null* both return the position of the first non-zero element of the input array by using two different algorithms. The program *find* returns the position p, with $p \in [0..n-1]$, of the first occurrence of a given value in the input array. If such position does not exist, it returns the value -1. Programs *init* and *init-partial* initialize to a constant the entire input array or a portion of it, respectively (the initialization starts from the first element). Program *init-backward* initializes to zero the entire input array, starting 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. The program *max* computes the maximum element of the input array. The program *partition* copies the non-negative and negative elements of the input array into two distinct arrays. The 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.

The source code of all these programs, for which we had to verify the properties indicated in the last column of Table 1, is available at `http://map.uniroma2.it/smc/arrays/`.

In order to verify the above programs we have applied the *Transform* strategy using different generalization operators, which are based on widening and convex hull. In particular, the Gen_W operator uses widening, and the Gen_{CHWS} operator uses a combination of widening and convex hull. The interested reader my refer to [17] for details on these operators. We have also combined these operators with a delay mechanism that, before starting the actual generalization process, introduces a definition which is computed by using convex hull alone, without widening. We denote Gen_{WD} and Gen_{CHWSD}, respectively, the operators Gen_W and Gen_{CHWS} combined with delay.

Example	Array program: *prog*	Property to be verified: $\neg\varphi_{error}$
bubblesort-inner	`for(j=0; j<n-i-1; j++) { if(a[j]>a[j+1]) { tmp=a[j]; a[j]=a[j+1]; a[j+1]=tmp; } }`	$\forall k.\ (0 \leq i < n \ \wedge\ 0 \leq k < j \ \wedge\ j = n-i-1) \rightarrow a[k] \leq a[j]$
insertionsort-inner	`x=a[i]; j=i-1; while(j>=0 && a[j]>x) { a[j+1]=a[j]; --j; }`	$\forall k.\ (0 \leq i < n \ \wedge\ j+1 < k \leq i) \rightarrow a[k] > x$
selectionsort-inner	`for(j=i+1; j<n; j++) { if(a[i]>a[j]) { tmp=a[i]; a[i]=a[j]; a[j]=tmp; } }`	$\forall k.(0 \leq i \leq k < n) \rightarrow a[k] \geq a[i]$
copy	`for(i=0; i<n; i++) a[i]=b[i];`	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = b[i]$
copy-partial	`for(i=0; i<k; i++) a[i]=b[i];`	$\forall i.\ (0 \leq i < k \leq n) \rightarrow a[i] = b[i]$
copy-reverse	`for(i=0;i<n;i++) b[i]=a[i]; for(i=0;i<n;i++) a[i]=b[n-i-1];`	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = b[n-i-1]$
difference	`for(i=0; i<n; i++) c[i]=a[i]-b[i];`	$\forall i.\ (0 \leq i < n) \rightarrow c[i] = a[i] - b[i]$
sum	`for(i=0; i<n; i++) c[i]=a[i]+b[i];`	$\forall i.\ (0 \leq i < n) \rightarrow c[i] = a[i] + b[i]$
find-first-non-null	`p=-1; for(i=0; i<n; i++) if(a[i]!=0) { p=i; break; }`	$(0 \leq p < n) \rightarrow a[p] \neq 0$
first-non-null	`s=n; for(i=0; i<n; ++i) if(s==n && a[i]!=0) s=i;`	$(0 \leq s < n) \rightarrow (\ a[s] \neq 0 \ \wedge$ $(\forall i.\ (0 \leq i < s) \rightarrow a[i] = 0)\ )$
find	`p=-1; for(i=0; i<n; i++) if(a[i]==e) { p=i; break; }`	$(0 \leq p < n) \rightarrow a[p] = e$
init	`for(i=0; i<n; i++) a[i]=c;`	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = c$
init-partial	`for(i=0; i<k; i++) a[i]=0;`	$\forall i.\ (0 \leq i < k \leq n) \rightarrow a[i] = 0$
init-backward	`i=n; while(i>0) { i=i-1; a[i]=0; }`	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = c$
init-non-constant	`for(i=0; i<n; i++) a[i]=2*i+c;`	$\forall i.\ (0 \leq i < n) \rightarrow a[i] = 2\,i + c$
init-sequence	`a[0]=7; i=1; while(i<n) { a[i]=a[i-1]+1; i++; }`	$\forall i.\ (1 \leq i < n) \rightarrow a[i] = a[i-1] + 1$
max	`m=a[0]; i=1; while(i<n) { if(a[i]>m) m=a[i]; i++; }`	$\forall i.\ (0 \leq i < n) \rightarrow m \geq a[i]$
partition	`i=0; j=0; k=0; while(i<n) { if(a[i]>=0) { b[j]=a[i]; j++;} else { c[k]=a[i]; k++; } ++i; }`	$(\forall i.\ (0 \leq i < j) \rightarrow b[i] \geq 0)\ \wedge$ $(\forall i.\ (0 \leq i < k) \rightarrow c[i] < 0)$
rearrange-in-situ	`i=0; j=n; while (i<j) { if (a[i]>=0) { i=i+1; } else { j=j-1; x=a[i]; a[i]=a[j]; a[j]=x; }}`	$(\forall k.(0 \leq k < i) \rightarrow a[k] \geq 0)\ \wedge$ $(\forall k.(j < k < n) \rightarrow a[k] < 0)$

Table 1. Benchmark array programs. `a`, `b`, `c` are integer arrays of size `n`. For all these examples φ_{init} is *true*.

In Table 2 we report the results obtained by applying the *Transform* strategy with the four generalization operators mentioned above. Column 'References' contains references to papers where the program verification example of that row has been considered.

The last four columns are labeled with the name of the generalization operator. For each program that has been proved correct, we report the time (in seconds) taken to verify the property of interest (see Table 1). By '*unknown*' we indicate that the *Transform* strategy derives a CLP(Array) program containing constrained facts different from '`incorrect`', and hence correctness (or incorrectness) of the imperative program cannot be proved.

Example	References	Gen_W	Gen_{WD}	Gen_{CHWS}	Gen_{CHWSD}
bubblesort-inner		0.67	0.85	0.70	0.88
insertionsort-inner	[25, 32, 45]	0.30	0.32	0.53	0.55
selectionsort-inner	[45]	*unknown*	1.34	1.16	1.38
copy	[5, 14, 25, 32, 45]	*unknown*	0.30	0.42	0.37
copy-partial	[5, 14]	*unknown*	0.33	0.42	0.34
copy-reverse	[5, 14]	*unknown*	0.36	0.68	0.63
difference	[5]	*unknown*	0.61	1.22	1.08
sum		*unknown*	0.65	1.30	1.13
find-first-non-null	[5, 14]	0.14	0.15	0.18	0.17
first-non-null	[25]	0.22	0.24	0.24	0.25
find	[5, 14]	0.33	0.49	0.58	0.53
init	[5, 14, 45]	*unknown*	0.15	0.20	0.19
init-partial	[5, 14]	*unknown*	0.13	0.21	0.16
init-backward	[8]	*unknown*	0.11	0.24	0.21
init-non-constant	[5, 14, 32, 45]	*unknown*	0.16	0.40	0.35
init-sequence	[25, 32]	*unknown*	0.63	0.93	0.85
max	[25, 32]	*unknown*	0.30	0.30	0.34
partition	[14, 32, 45]	0.49	0.53	0.56	0.55
rearrange-in-situ	[8]	*unknown*	*unknown*	0.79	0.86
	precision	6	18	19	19
	total time	2.15	7.65	11.06	10.82
	average time	0.36	0.42	0.58	0.57

Table 2. Verification results using VeriMAP with different generalization operators. Times are in seconds.

We also report, for each generalization operator, the number of successfully verified programs (which measures the *precision* of the operator), the *total time* taken to perform all the benchmark verification examples, and the *average time* per successful answer.

All experiments have been done on a single core of 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 by using the Gen_W operator, which is based on widening alone, our method is able to prove only 6 programs out of 19. However, precision can be increased by

using the operators Gen_{WD}, Gen_{CHWS}, and Gen_{CHWSD}, 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 set of examples) only a slight increase of verification times.

A detailed comparison of the performance of our verification system and the proposed *Transform* strategy with respect to the other available verification systems, whose references can be found in the papers cited in Table 2, is difficult to make at this time, because those systems are not all readily available and also the results reported in the literature do not refer to the same code for the input C programs.

7. Related Work and Conclusions

The verification method presented in this paper is an extension of the one introduced in [10], where programs that manipulate arrays were not considered. Some examples suggesting how arrays and also recursively defined properties can be dealt with in our transformational approach, were presented in [11], where, however, no automatic strategy was presented. In this paper, that extends [12], we have shown that by applying a rather simple and general, automated transformation strategy, it is possible to do most of the verification examples found in the literature, with a good time performance. We are currently working on the extension of our strategy to deal with recursive array programs such as *quicksort*.

The idea of encoding imperative programs into CLP programs for reasoning about their properties was presented in various papers [18, 28, 40], 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 with respect to [18, 28, 40] is that we use CLP *program transformations* to prove properties, rather than symbolic execution or static analysis.

The verification method presented in this paper is also related to several other methods that use abstract interpretation and theorem proving techniques and now we briefly report on those methods.

Among the papers that use abstract interpretations for finding invariants of programs that manipulate arrays, let us recall [25]. In that paper, which builds upon [21], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is followed in [8] where a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs is introduced. In [19, 31] 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.

Methods based on abstract interpretation construct overapproximations of the behavior 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 finite 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 [3, 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 [29, 30, 36] the authors present some techniques based on theorem proving which may generate array invariants. In [45] a backward reachability analysis based on predicate abstraction

and abstraction refinement is used for verifying assertions that are universally quantified over array indexes. Finally, we would like to mention that techniques based on Satisfiability Modulo Theory (SMT) have been used for generating and verifying universally quantified properties over array variables (see, for instance, [2, 32]).

The approaches based on theorem proving and SMT are more flexible with respect to those based on abstract interpretation, because no finite 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.

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 (such as [10, 17, 40]) show that within that transformation approach 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 program syntax and semantics, 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, (i) optimizing transformations can be applied to improve the efficiency of the verification task, and (ii) transformations can be composed together to derive sophisticated verification techniques. For instance, in [10] it is shown that the *iteration* of program specialization combined with suitable constraint propagations can significantly improve the precision of our program verification method.

In order to further validate our approach, we plan for the future to address the issue of proving correctness of programs manipulating *dynamic data structures* such as lists or heaps, looking for a set of suitable 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 Satisfiability Modulo Theory solvers. An interesting direction for future research is also the combination of semantics preserving transformations of CLP programs (like the ones considered in this paper) with other techniques that use a CLP representation of the verification conditions.

Acknowledgements

We would like to thank the anonymous referees of CILC 2013 for their helpful comments and constructive criticism.

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, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *CAV '12*, LNCS 7358, pages 679–685. Springer, 2012.

[3] F. Alberti, S. Ghilardi, and N. Sharygina. Decision Procedures for Flat Array Properties. *Proc. TACAS '14*, LNCS 8413, pages 15–30. 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 Programs via Iterated Specialization. In *PEPM '13*, pages 43–52, ACM, 2013.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verification of Imperative Programs by Constraint Logic Program Transformation. In *SAIRP '13, Festschrift for Dave Schmidt*, Electronic Proceedings in Theoretical Computer Science, Vol. 129, pages 186–210, 2013.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Array Programs by Transforming Verification Conditions. Proc. 15th International Conference on Verification, Model Checking, and Abstract Interpretation. *VMCAI '14*, LNCS 8318, pages 182–202. Springer, 2014.

[13] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. Proc. 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems *TACAS '14*, LNCS 8413, pages 568–574. Springer, 2014.

[14] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *ESOP'10*, 2010.

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

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

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

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

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

[27] J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification. http://paella.d1.comp.nus.edu.sg/tracer/, 2012.

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

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

[30] 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.

[31] S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.

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

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

[34] The MAP transformation system. http://www.iasi.cnr.it/∼proietti/system.html. Also available via a WEB interface from `http://www.map.uniroma2.it/mapweb`.

[35] 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.

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

[37] 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.

[38] 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.

[39] 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.

[40] 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.

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

[42] 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.

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

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

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