Verifying Array Programs by Transforming Verification Conditions

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

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

Abstract. We present a method for verifying properties of imperative programs manipulating integer arrays. We assume that we are given a program and a property to be verified. The *interpreter* (that is, the operational semantics) of the program is specified as a set of Horn clauses with constraints in the domain of integer arrays, also called *constraint logic programs over integer arrays*, denoted CLP(Array). Then, by specializing the interpreter with respect to the given program and property, we generate a set of *verification conditions* (expressed as a CLP(Array) program) whose satisfiability implies that the program verifies the given property. Our verification method is based on transformations that preserve the least model semantics of CLP(Array) programs, and hence the satisfiability of the verification conditions. In particular, we apply the usual rules for CLP transformation, such as unfolding, folding, and constraint replacement, tailored to the specific domain of integer arrays. We propose an automatic strategy that guides the application of those rules with the objective of deriving a new set of verification conditions which is either trivially satisfiable (because it contains no constrained facts) or is trivially unsatisfiable (because it contains the fact *false*). Our approach provides a very rich program verification framework where one can compose together several verification strategies, each of them being implemented by transformations of CLP(Array) programs.

1 Introduction

Horn clauses and constraints have been advocated by many researchers as suitable logical formalisms for the automated verification of imperative programs [2, 19, 34]. Indeed, the *verification conditions* that express the correctness of a given program, can often be expressed as *constrained Horn clauses* [3], that is, Horn clauses extended with constraints in specific domains such as the integers or the rationals. For instance, consider the following C-like program *prog*:

$x{=}0;\ \ y{=}0;$
`while` $(x{<}n)\ \{x{=}x{+}1;\ y{=}y{+}2\}$

and assume that we want to prove the following Hoare triple: $\{n \geq 1\}\ prog\ \{y > x\}$. This triple is valid if we find a predicate P such that the following three verification conditions hold:

1. $x{=}0 \wedge y{=}0 \wedge n{\geq}1 \rightarrow P(x, y, n)$
2. $P(x, y, n) \wedge x{<}n \rightarrow P(x+1, y+2, n)$
3. $P(x, y, n) \wedge x{\geq}n \rightarrow y{>}x$

Constraints such as the equalities and inequalities in clauses 1–3, are formulas defined in a background (possibly non-Horn) theory. The use of constraints makes it easier to express the properties of interest and enables us to apply ad-hoc theorem provers, or *solvers*, for reasoning over those properties.

Verification conditions can be automatically generated either from a formal specification of the operational semantics of the programs [34] or from the proof rules that formalize program correctness in an axiomatic way [19].

The correctness of a program is implied by the satisfiability of the verification conditions. Various methods and tools for *Satisfiability Modulo Theory* (see, for instance, [11]) prove the correctness of a given program by finding an interpretation (that is, a relation specified by constraints) that makes the verification conditions true. For instance, in our example, one such interpretation is:

$P(x, y, n) \equiv (x{=}0 \wedge y{=}0 \wedge n{\geq}1) \vee y{>}x$

It has been noted (see, for instance, [3]) that verification conditions can be viewed as *constraint logic programs*, also called *CLP programs* [22]. Indeed, clauses 1 and 2 above can be considered as clauses of a CLP program over the integers, and clause 3 can be rewritten as the following *goal* (by moving the conclusion to the premises):

4. $P(x, y, n) \wedge x{\geq}n \wedge y{\leq}x \rightarrow \mathit{false}$

Various verification methods based on constraint logic programming have been proposed in the literature (see, for instance, [8, 10, 34]). These methods consist of two steps: (i) the first step is the translation of the verification task into a CLP program, and (ii) the second step is the analysis of that CLP program. In particular, as indicated in [8], in many cases it is helpful for the analysis step to transform a CLP program (expressing a set of verification conditions) into an equisatisfiable program whose satisfiability is easier to show.

For instance, if we propagate, according to the transformations described in [8], the two constraints representing the initialization condition ($x{=}0 \wedge y{=}0 \wedge n{\geq}1$) and the error condition ($x{\geq}n \wedge y{\leq}x$), then from clauses 1, 2, and 4 we derive the following new verification conditions:

5. $Q(x, y, n) \wedge x{<}n \wedge x{>}y \wedge y{\geq}0 \rightarrow Q(x+1, y+2, n)$
6. $Q(x, y, n) \wedge x{\geq}n \wedge x{\geq}y \wedge y{\geq}0 \wedge n{\geq}1 \rightarrow \mathit{false}$

This propagation of constraints preserves the least model, and hence, by extending the van Emden-Kowalski Theorem [38] to constrained Horn clauses, the verification conditions expressed by clauses 5–6 are satisfiable iff clauses 1–3 are satisfiable. Now, proving the satisfiability of clauses 5–6 is trivial because none of them is a *constrained fact* (that is, a clause of the form $c \rightarrow Q(x, y, n)$, where c is a satisfiable constraint). Thus, clauses 5-6 are made true by simply taking $Q(x, y, n)$ to be *false*.

The approach presented in [8] shows that the transformational verification method briefly presented in the example above, is quite general. According to that method, in fact, one starts from a program *prog* on integers and a safety

property φ to be verified. Then, following [34], one specifies the *interpreter* of the program as a CLP program whose constraints are in the domain of integer arrays. Next, by specializing the interpreter with respect to *prog* and φ, a new CLP program, call it VC, is derived. This program consists of the clauses that express the verification conditions (hence the name VC) which guarantee that *prog* satisfies φ. Program VC (and the corresponding set of verification conditions) is repeatedly specialized with respect to the constraints occurring in its clauses with the objective of deriving either (i) a CLP program without constrained facts, hence proving that *prog* satisfies φ, or (ii) a CLP program containing the fact *false*, hence proving that *prog* does not satisfy φ (in this case a counterexample to φ can be extracted from the derivation of the specialized program).

In this paper we extend the method presented in [8] to the proof of partial correctness properties of programs manipulating integer arrays. In order to specify verification conditions for array programs, in Section 2 we introduce the class of CLP(Array) programs, that is, logic programs with constraints in the domain of integer arrays. In particular, CLP(Array) programs may contain occurrences of `read` and `write` predicates that are interpreted as the input and output relations of the usual read and write operations on 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 [4, 17, 30]. In Section 4 we show how to generate the verification conditions via 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. In particular, similarly to [8], the strategy aims at deriving either (i) a CLP(Array) program that has no constrained facts (hence proving satisfiability of the verification conditions and partial correctness of the program), or (ii) a CLP(Array) program containing the fact *false* (hence proving that the verification conditions are unsatisfiable and the program does not satisfy the given property). 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 [14]. Finally, as reported in Section 6, we have implemented our transformation strategy on the MAP transformation system [29] and we have tested the verification method using the strategy we have proposed on a set of array programs taken from the literature.

2 Constraint Logic Programs on Arrays

In this section we recall some basic notions and terminology concerning Constraint Logic Programming (CLP), and we introduce the set CLP(Array) of CLP programs with constraints in the domain of integer arrays. For details on CLP the reader may refer to [22].

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*. The dimension `n` of

an array a is represented as a binary relation by the predicate dim(a, n). For reasons of simplicity we consider one-dimensional arrays only. The read and write operations on arrays are represented by the predicates read and write, respectively, as follows: read(a, i, v) denotes that the i-th element of array a is the value v, and write(a, i, v, b) denotes that the array b that is equal to the array a except that its i-th element is v. We assume that both indexes and values are integers, but our method is parametric with respect to the index and value domains. (Note, however, that the result of a verification task may depend on the constraint solver used, and hence on the constraint domain.)

An *atomic array constraint* is an atom of the following form: either dim(a, n), or read(a, i, v), or write(a, i, v, 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 $\mathtt{t}_1, \ldots, \mathtt{t}_m$ 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. A is called the *head* and c, B is called the *body* of the clause. We assume that in every clause all integer arguments in its head are distinct variables. The clause A :- c is called a *constrained fact*. When 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. We say that a predicate p in a linear program P is *useless* if in P there are constrained facts neither for p nor for each predicate q on which q depends.

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

(i) the set D is the Herbrand universe [28] constructed out of the set $\mathbb{Z}$ of the integers, the constants, and the function symbols different from + and *,

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

(iii) for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{n-1}$, for all integers d, $\mathtt{dim}(\mathtt{a}_0 \ldots \mathtt{a}_{n-1}, \mathtt{d})$ is true in I iff $\mathtt{d} = \mathtt{n}$

(iv) I interprets the predicates read and write as follows: for all sequences $\mathtt{a}_0 \ldots \mathtt{a}_{n-1}$ and $\mathtt{b}_0 \ldots \mathtt{b}_{m-1}$ of integers, for all integers i and v, $\mathtt{read}(\mathtt{a}_0 \ldots \mathtt{a}_{n-1}, \mathtt{i}, \mathtt{v})$ is true in I iff $0 \leq \mathtt{i} \leq \mathtt{n}-1$ and $\mathtt{v} = \mathtt{a}_i$, and $\mathtt{write}(\mathtt{a}_0 \ldots \mathtt{a}_{n-1}, \mathtt{i}, \mathtt{v}, \mathtt{b}_0 \ldots \mathtt{b}_{m-1})$ is true in I iff
$0 \leq \mathtt{i} \leq \mathtt{n}-1$, $\mathtt{n} = \mathtt{m}$, $\mathtt{b}_i = \mathtt{v}$, and for $\mathtt{j} = 0, \ldots, \mathtt{n}-1$, if $\mathtt{j} \neq \mathtt{i}$ then $\mathtt{a}_j = \mathtt{b}_j$

(v) I is an Herbrand interpretation [28] for function and predicate symbols different from $+, *, =, \geq, >$, dim, read, and write.

We can identify an $\mathcal{A}$-interpretation I with the set of ground atoms that are true in I, and hence $\mathcal{A}$-interpretations are partially ordered by set inclusion.

We write $\mathcal{A} \models \varphi$ if φ is true in every $\mathcal{A}$-interpretation. A constraint $\mathtt{c}$ is *satisfiable* if $\mathcal{A} \models \exists(\mathtt{c})$, where in general, 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 $\mathtt{c}$ *entails* a constraint $\mathtt{d}$, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathcal{A} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$. By $vars(\varphi)$ we denote the free variables of φ.

We assume that we are given a solver to check the satisfiability and the entailment of constrains in $\mathcal{A}$. To this aim we can use any solver that implements algorithms for satisfiability and entailment in the theory of integer arrays [4, 17].

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.

Given a CLP(Array) program P and a ground goal $\mathtt{G}$ of the form $\mathtt{:-A}$, $P \cup \{\mathtt{G}\}$ is satisfiable (or, equivalently, $P \not\models \mathtt{A}$) if and only if $\mathtt{A} \notin M(P)$. This property is a straightforward extension to CLP(Array) programs of van Emden and Kowalski's result [38].

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, [13]).

Let P be a CLP(Array) program.

Definition Rule. By this 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 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. Given 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 consider the set $\{\mathtt{K}_i \mathtt{ :- c}_i\mathtt{,B}_i \mid i = 1, \ldots, m\}$ made out 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 $(\mathtt{c,c}_i)\,\vartheta_i$ is satisfiable. By unfolding C w.r.t. A using P, we derive the set $\{(\mathtt{H :- c,c}_i\mathtt{,L,B}_i\mathtt{,R})\,\vartheta_i \mid i = 1, \ldots, m\}$ of clauses.

Constraint Replacement Rule. If a constraint $\mathtt{c_0}$ occurs in the body of a clause C and, for some constraints $\mathtt{c_1}, \ldots, \mathtt{c_n}$,

$\mathcal{A} \models \forall\,((\exists \mathtt{X_0}\ \mathtt{c_0}) \leftrightarrow (\exists \mathtt{X_1}\ \mathtt{c_1} \vee \ldots \vee \exists \mathtt{X_n}\ \mathtt{c_n}))$

where, for $i = 0, \ldots, n$, $\mathtt{X_i} = vars(C) - vars(\mathtt{c_i})$, then we derive n new clauses $C_1, \ldots, C_n$ by replacing $\mathtt{c_0}$ by $\mathtt{c_1}, \ldots, \mathtt{c_n}$, respectively, in the body of C.

The equivalences needed for constraint replacements are shown to hold in $\mathcal{A}$ by using a relational version of the theory of arrays with dimension [4, 17]. In particular, the constraint replacements we apply during the transformations

described in Section 5 follow from the following axioms where all variables are universally quantified at the front:

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

(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 omit the usual axioms for `dim`.

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 `H :- e, L,` $\mathtt{K}\,\vartheta$`, 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 correctness results for the unfold/fold transformations of CLP programs proved in [13] can be instantiated to our context as stated in the following theorem.

Theorem 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 (see the details in [13]). Informally, this assumption avoids the replacement of a definition clause `A :- B` with the clause `A :- A` obtained by folding `A :- B` using itself. This replacement may not preserve the least model semantics.

4 Generating Verification Conditions via Specialization

We consider an imperative C-like 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, and in each program there is a unique `halt` command which, when executed, causes program termination.

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 [36], and is omitted.

Given an imperative program *prog*, we address the problem of verifying whether or not, starting from any *initial configuration* that satisfies the property φ_{init}, the execution of *prog* eventually leads to a *final configuration* that satisfies the property φ_{error}, also called an *error configuration*. This problem is formalized by defining an *incorrectness triple* of the form $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$, where φ_{init} and φ_{error} are constraints. We say that a program *prog* is *incorrect* with respect to φ_{init} and φ_{error}, whose free variables are assumed to be among the program variables $z_1, \ldots, z_r$, if there exist environments δ_{init} and δ_h such that: (i) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds, (ii) $\langle\!\langle \ell_0 : c_0,\ \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \texttt{halt},\ \delta_h \rangle\!\rangle$, and (iii) $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds, where $\ell_0 : c_0$ is the first labeled command of *prog* and ℓ_h : `halt` is the unique `halt` command of *prog*. A program is said to be *correct* with respect to φ_{init} and φ_{error} iff it is not incorrect with respect to φ_{init} and φ_{error}. Note that our notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$. In this paper we assume that the properties φ_{init} and φ_{error} can be expressed as conjunctions of (integer and array) constraints.

We translate the problem of checking whether or not the program *prog* is incorrect with respect to the properties φ_{init} and φ_{error} into the problem of checking whether or not the nullary predicate `incorrect` (standing for *false*) is a consequence of the CLP(Array) program T 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)`. Those clauses are defined as follows: (i) `initConf(X)` encodes an initial configuration satisfying the property φ_{init}, (ii) `errorConf(X)` encodes an error configuration satisfying the property φ_{error}, and (iii) `tr(X,Y)` encodes the transition relation $\Longrightarrow$ between pairs of configurations, which depends on the given program *prog*. For instance, the following clause encodes the transition relation for the array assignment $\ell : a[ie] = e$ (here a configuration pair of the form: $\langle\!\langle \ell : c, \delta \rangle\!\rangle$ for the command c at label ℓ and the environment δ, 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).
```

(`L1` is the label following `L` in the encoding of the given program.) The predicate `reach(Y)` holds if a configuration `Y` can be reached from an initial configuration.

The imperative program *prog* is correct with respect to the properties φ_{init} and φ_{error} iff `incorrect` $\notin M(T)$ (or, equivalently, $T \not\models$ `incorrect`), where $M(T)$ is the least $\mathcal{A}$-model of program T (see Section 2). Our verification method performs a sequence of applications of the unfold/fold rules presented in Section 3 starting from program T. By Theorem 1 we have that, for each program U obtained from T by a sequence of applications of the rules, `incorrect` $\in M(T)$ iff `incorrect` $\in M(U)$.

Our verification method is made out of the following two steps, each of which is realized by a sequence of applications of the unfold/fold transformation rules: *Step* (A): *Generation of Verification Conditions*, and *Step* (B): *Transformation of Verification Conditions.*

In Step (A) program T is *specialized* with respect to the given `tr` (which depends on *prog*), `initConf`, and `errorConf`, thereby deriving a new program $T1$ such that: (i) $\texttt{incorrect} \in M(T)$ iff $\texttt{incorrect} \in M(T1)$, and (ii) `tr` does not occur explicitly in $T1$. The specialization of T is obtained by applying a variant of the strategy for interpreter removal presented in [8]. The main difference with respect to [8] is that the CLP programs considered in this paper contain `read`, `write`, and `dim` predicates. The `read` and `write` predicates are never unfolded during specialization and they occur in the residual CLP(Array) program $T1$. All occurrences of the `dim` predicate are eliminated by replacing them by suitable integer constraints on indexes. The clauses of $T1$ are called the *verification conditions* for *prog*, and we say that they are *satisfiable* iff $\texttt{incorrect} \notin M(T1)$ (or equivalently $T1 \not\models \texttt{incorrect}$). Thus, the satisfiability of the verification conditions for *prog* guarantees that *prog* is correct with respect to φ_{init} and φ_{error}.

Step (B) has the objective of checking, through further transformations, the satisfiability of the verification conditions generated by Step (A). We will describe this step in detail in Section 5.

Let us consider, for example, 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: $\ell_0\colon\ i = 1;$
$\ell_1\colon$ `while` $(i{<}n)\ \{\ a[i] = a[i{-}1] + 1;\ i = i + 1;\ \};$
$\ell_h\colon$ `halt`

We consider the following incorrectness triple:

$\{\!\{\varphi_{init}(i,n,a)\}\!\}\ \mathit{SeqInit}\ \{\!\{\varphi_{error}(n,a)\}\!\}$

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 $\exists j\ (0{\leq}j\ \wedge\ j+1{<}n\ \wedge\ a[j]{\geq}a[j{+}1])$.

First, the above incorrectness triple is translated into a CLP(Array) program T. 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) :- Z=W+1, W≥0, W+1<N, U≥V, read(A,W,U), read(A,Z,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),Ps)):-at(l0,Cmd), progState(Ps), phiInit(Ps).`
4. `errorConf(cf(cmd(lh,Cmd),Ps)):-at(lh,Cmd), progState(Ps), phiError(Ps).`

The predicates `at` and `progState` are defined by: '`at(l0, asgn(int(i), int(1))).`', '`at(lh, halt).`', and '`progState([[int(i), I], [int(n), N], [array(a), A]]).`'.

Now we apply Step (A) of our verification method, which consists in the removal of the interpreter. From program T we obtain the following program $T1$:

5. `incorrect :- Z=W+1, W≥0, W+1<N, U≥V, N≤I, read(A,W,U), read(A,Z,V), p(I,N,A).`
6. `p(I1,N,B) :- 1≤I, I<N, D=I−1, I1=I+1, V=U+1, read(A,D,U), write(A,I,V,B), p(I,N,A).`
7. `p(I,N,A) :- I=1, N≥1.`

The CLP(Array) program $T1$ expresses the verification conditions for *SeqInit*. Indeed, predicate `p` is an invariant for the `while` loop. For reasons of simplicity, the predicates expressing the assertions associated with assignments and conditionals have been unfolded away during the removal of the interpreter. (The strategy for removing the interpreter can be customized.)

Due to the presence of integer and array variables, the least $\mathcal{A}$-model $M(T1)$ 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). Thus, we cannot directly use the standard CLP systems to prove program correctness. In order to cope with this difficulty, we use a method based on CLP program transformations, which allows us to avoid the exhaustive exploration of the possibly infinite space of reachable configurations.

5 A Transformation Strategy for Verification

As mentioned above, the verification conditions expressed as the CLP(Array) program $T1$ generated by Step (A) are satisfiable iff $\texttt{incorrect} \notin M(T1)$. Our verification method is based on the fact that by transforming the CLP(Array) program $T1$ using rules that preserve the least $\mathcal{A}$-model, we get a new CLP(Array) program $T2$ that expresses equisatisfiable verification conditions.

Step (B) has the objective of showing, through further transformations, that *either* the verification conditions generated by Step (A) are satisfiable (that is, $\texttt{incorrect} \notin M(T1)$ and hence *prog* is correct with respect to φ_{init} and φ_{error}), *or* they are unsatisfiable (that is, $\texttt{incorrect} \in M(T1)$ and hence *prog* is not correct with respect to φ_{init} and φ_{error}). To this aim, Step (B) propagates the initial and/or the error properties so as to derive from program $T1$ a program $T2$ where the predicate `incorrect` is defined by either (α) the fact '`incorrect`' (in which case the verification conditions are unsatisfiable and *prog* is incorrect), or (β) the empty set of clauses (in which case the verification conditions are satisfiable and *prog* is correct). In the case where neither (α) nor (β) holds, that is, in program $T2$ the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact '`incorrect`', we cannot conclude anything about the correctness of *prog*. However, similarly to what has been proposed in [8], in this case we can iterate Step (B), alternating the propagation of the initial and error properties, in the hope of deriving a program where either (α) or (β) holds. Obviously, due to undecidability limitations, it may be the case that we never get a program where either (α) or (β) holds.

Step (B) 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 error property. The forward propagation of the initial property can be obtained by combining *Transform* with the *Reversal*

transformation described in [8]. For lack of space we do not present this extra transformation here.

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

INITIALIZATION:
Let *InDefs* be the set of all clauses of $T1$ whose head is the atom `incorrect`;
$T2 := \emptyset$; *Defs* := *InDefs*;

while in *InDefs* there is a clause C *do*

UNFOLDING: Unfold C w.r.t. the unique atom in its body using $T1$, and derive a set $U(C)$ of clauses;

CONSTRAINT REPLACEMENT: Apply a sequence of constraint replacements by using 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* $- \{C\}$; $T2 := T2 \cup F(C)$;

end-while;

REMOVAL OF USELESS CLAUSES:
Remove from $T2$ all clauses with head predicate `p`, if in $T2$ there is no constrained fact `q`$(\ldots)$ `:- c` where `q` is either `p` or a predicate on which `p` depends.

Fig. 1. The *Transform* strategy.

The input program $T1$ is a *linear* CLP(Array) program (we can show, in fact, that Step (A) always generates a linear program).
UNFOLDING performs one inference step backward from the error property.
The CONSTRAINT REPLACEMENT phase by applying the theory of arrays, infers new constraints on the variables of the only atom that occurs in the body of each clause obtained by the UNFOLDING phase. It works as follows. We select a clause, say `H :- c, G`, in the set $U(C)$ of the clauses obtained by unfolding, and we replace that clause by the one(s) obtained by applying as long as possible the following rules. Note that this process always terminates and, in general, it is nondeterministic.

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

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

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

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

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).

The DEFINITION & FOLDING phase introduces new predicate definitions by suitable generalizations of the constraints. These generalizations guarantee the termination of *Transform*, but at the same time they should be as specific as possible in order to achieve maximal precision. This phase works as follows. Let $C1$ in $R(C)$ be a clause of the form $\mathtt{H}$:- $\mathtt{c}, \mathtt{p(X)}$. 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: $\mathtt{newp(X)}$:- $\mathtt{d}, \mathtt{p(X)}$ such that $\mathit{vars}(\mathtt{d}) \subseteq \mathit{vars}(\mathtt{c})$ and $\mathtt{c} \sqsubseteq \mathtt{d}$, then we fold $C1$ using D. Otherwise, we introduce a clause of the form $\mathtt{newp(X)}$:- $\mathtt{gen}, \mathtt{p(X)}$ where: (i) $\mathtt{newp}$ is a predicate symbol occurring neither in the initial program nor in *Defs*, and (ii) $\mathtt{gen}$ is a constraint such that $\mathit{vars}(\mathtt{gen}) \subseteq \mathit{vars}(\mathtt{c})$ and $\mathtt{c} \sqsubseteq \mathtt{gen}$. The constraint $\mathtt{gen}$ is called a *generalization* of the constraint $\mathtt{c}$ and is constructed as follows.

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 $\mathtt{read}$ and $\mathtt{write}$ constraints.
(1) Delete all $\mathtt{write}$ constraints from $\mathtt{rw}_1$, hence deriving $\mathtt{r}_1$.
(2) Rewrite $\mathtt{i}_1, \mathtt{r}_1$ so that all occurrences of integers in $\mathtt{read}$ constraints are distinct variables not appearing in $\mathtt{X}$ (this can be achieved by possibly adding some integer equalities to $\mathtt{r}_1$), hence deriving $\mathtt{i}_2, \mathtt{r}_2$.
(3) Compute the projection $\mathtt{i}_3$ (in the rationals) of the constraint $\mathtt{i}_2$ onto $\mathit{vars}(\mathtt{r}_2) \cup \{\mathtt{X}\}$ (thus $\mathtt{i}_2 \sqsubseteq \mathtt{i}_3$ in the domain of the integers).
(4) Delete from $\mathtt{r}_2$ 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 $\mathtt{i}_3$, thereby deriving a new value for $\mathtt{r}_2$. If at least one $\mathtt{read}$ has been deleted from $\mathtt{r}_2$ then go to step (3).
Let $\mathtt{i}_3, \mathtt{r}_3$ be the constraint obtained after the applications of steps (3)–(4).
(5) *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 $\mathtt{r}_3, \mathtt{p(X)}$,
Then compute a generalization $\mathtt{g}$ of the constraints $\mathtt{i}_3$ and $\mathtt{i}_0$ such that $\mathtt{i}_3 \sqsubseteq \mathtt{g}$, by using a *generalization operator* for linear constraints (we refer to [7, 14, 33] 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 $\mathtt{i}_3, \mathtt{r}_3$.

The correctness of the strategy with respect to the least $\mathcal{A}$-model semantics follows from Theorem 1, by observing that every clause defining a new predicate

introduced by DEFINITION & FOLDING is unfolded once during the execution of the strategy (indeed every such clause is added to *InDefs*).

The termination of the *Transform* strategy is based on the following facts:
(i) Constraint satisfiability and entailment are checked by a terminating solver (note that completeness is not necessary for the termination of *Transform*).
(ii) CONSTRAINT REPLACEMENT terminates (see above).
(iii) The set of new clauses that, during the execution of the *Transform* strategy, can be introduced by DEFINITION & FOLDING steps is finite. Indeed, by construction, they are all of the form $\mathtt{H}$:- $\mathtt{i}, \mathtt{r}, \mathtt{p}(\mathtt{X})$, where: (1) $\mathtt{X}$ is a tuple of variables, (2) $\mathtt{i}$ is an integer constraint, (3) $\mathtt{r}$ is a conjunction of array constraints of the form $\mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{V})$, where $\mathtt{A}$ is a variable in $\mathtt{X}$ and the variables $\mathtt{I}$ and $\mathtt{V}$ occur in $\mathtt{i}$ only, (4) the cardinality of $\mathtt{r}$ is bounded, because generalization does not introduce a clause $\mathtt{newp}(\mathtt{X})$:- $\mathtt{i}_3, \mathtt{r}_3, \mathtt{p}(\mathtt{X})$ if there exists an ancestor clause 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 $\mathtt{r}_3, \mathtt{p}(\mathtt{X})$, (5) 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 [7, 14, 33] satisfy this finiteness property.

Theorem 2. (Termination and Correctness of the *Transform* strategy) (i) *The Transform strategy terminates.* (ii) *Let program $T2$ be the output of the Transform strategy applied on the input program $T1$. Then,* $\mathtt{incorrect} \in M(T1)$ *iff* $\mathtt{incorrect} \in M(T2)$.

Let us now consider again the *SeqInit* example of Section 4 and perform Step (B). We apply the *Transform* strategy starting from program $T1$.

UNFOLDING. First, we unfold clause 5 w.r.t. the atom $\mathtt{p}(\mathtt{I}, \mathtt{N}, \mathtt{A})$, and we get:

8. $\mathtt{incorrect}$:- $\mathtt{Z}{=}\mathtt{W}{+}1$, $\mathtt{W}{\geq}0$, $\mathtt{Z}{\leq}\mathtt{I}$, $\mathtt{D}{=}\mathtt{I}{-}1$, $\mathtt{N}{=}\mathtt{I}{+}1$, $\mathtt{Y}{=}\mathtt{X}{+}1$, $\mathtt{U}{\geq}\mathtt{V}$, $\mathtt{read}(\mathtt{B}, \mathtt{W}, \mathtt{U})$, $\mathtt{read}(\mathtt{B}, \mathtt{Z}, \mathtt{V})$, $\mathtt{read}(\mathtt{A}, \mathtt{D}, \mathtt{X})$, $\mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$, $\mathtt{p}(\mathtt{I}, \mathtt{N}, \mathtt{A})$.

CONSTRAINT REPLACEMENT. Then, by applying the replacement rules WR2, WR3, and RR1 to clause 8, we get the following clause:

9. $\mathtt{incorrect}$:- $\mathtt{Z}{=}\mathtt{W}{+}1$, $\mathtt{W}{\geq}0$, $\mathtt{Z}{<}\mathtt{I}$, $\mathtt{D}{=}\mathtt{I}{-}1$, $\mathtt{N}{=}\mathtt{I}{+}1$, $\mathtt{Y}{=}\mathtt{X}{+}1$, $\mathtt{U}{\geq}\mathtt{V}$, $\mathtt{read}(\mathtt{A}, \mathtt{W}, \mathtt{U})$, $\mathtt{read}(\mathtt{A}, \mathtt{Z}, \mathtt{V})$, $\mathtt{read}(\mathtt{A}, \mathtt{D}, \mathtt{X})$, $\mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$, $\mathtt{p}(\mathtt{I}, \mathtt{N}, \mathtt{A})$.

In particular, since $\mathtt{W} \neq \mathtt{I}$ is entailed by the constraint in clause 8, we apply rule WR2 and we obtain a new clause, say 8.1, where $\mathtt{read}(\mathtt{B}, \mathtt{W}, \mathtt{U}), \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$ is replaced by $\mathtt{read}(\mathtt{A}, \mathtt{W}, \mathtt{U}), \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$. Then, since neither $\mathtt{Z}{=}\mathtt{I}$ nor $\mathtt{Z}{\neq}\mathtt{I}$ is entailed by the constraint in clause 8.1, we apply rule WR3 and we obtain two clauses 8.2 and 8.3, where the constraint $\mathtt{read}(\mathtt{B}, \mathtt{Z}, \mathtt{V}), \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$ is replaced by $\mathtt{Z} = \mathtt{I}, \mathtt{Y} = \mathtt{V}, \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$ and $\mathtt{Z} \neq \mathtt{I}, \mathtt{read}(\mathtt{A}, \mathtt{Z}, \mathtt{U}), \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{Y}, \mathtt{B})$, respectively. Finally, since $\mathtt{D} = \mathtt{W}$ is entailed by the constraint in clause 8.2, we apply rule RR1 to clause 8.2 and we add the constraint $\mathtt{X} = \mathtt{U}$ to its body, hence deriving the unsatisfiable constraint $\mathtt{X} = \mathtt{U}, \mathtt{Y} = \mathtt{X} + 1, \mathtt{Y} = \mathtt{V}, \mathtt{U} \geq \mathtt{V}$. Thus, the clause derived by the latter replacement is removed. Clause 9 is derived from 8.3 by rewriting $\mathtt{Z} \leq \mathtt{I}, \mathtt{Z} \neq \mathtt{I}$ as $\mathtt{Z} < \mathtt{I}$.

DEFINITION & FOLDING. In order to fold clause 9 we introduce a new definition by applying Steps (1)–(5) of the DEFINITION & FOLDING phase. In particular, by deleting the `write` constraint (Step 1) and projecting the integer constraint (Step 3), we get a constraint where the variable `X` occurs in `read(A,D,X)` only. Thus, we delete `read(A,D,X)` (Step 4). Finally, we compute a generalization of the constraints occurring in clauses 5 and 9 by using the convex hull operator (Step 5). We get:

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

By folding clause 9 using clause 10, we get:

11. `incorrect :- Z=W+1, W≥0, Z<I, D=I−1, N=I+1, Y=X+1, U≥V, read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), new1(I,N,A).`

Now we proceed by performing a second iteration of the body of the while-loop of the *Transform* strategy because *InDefs* is not empty (indeed, at this point clause 10 belongs to *InDefs*).

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

12. `new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1, Z≤I, Z≥1, N>I, U≥V, read(B,W,U), read(B,Z,V), read(A,D,X), write(A,I,Y,B), p(I,N,A).`

CONSTRAINT REPLACEMENT. Then, by applying rules RR1, WR2, and WR3 to clause 12, we get the following clause:

13. `new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1, Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), p(I,N,A).`

DEFINITION & FOLDING. In order to fold clause 13 we introduce the following clause, whose body is derived by computing the widening $[5, 7]$ of the integer constraints in the ancestor clause 10 with respect to the integer constraints in clause 13:

14. `new2(I,N,A) :- Z=W+1, W≥0, W≤I−1, N>Z, U≥V, read(A,W,U), read(A,Z,V), p(I,N,A).`

By folding clause 13 using clause 14, we get:

15. `new1(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2, I≥1, Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), new2(I,N,A).`

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 unfolding and constraint replacement steps, followed by a final folding step, from clause 14 we get:

16. `new2(I1,N,B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, I≥1, Z<I, Z≥1, N>I, U≥V, read(A,W,U), read(A,Z,V), read(A,D,X), write(A,I,Y,B), new2(I,N,A).`

The final transformed program is made out of clauses 11, 15, and 16. Since this program has no constrained facts, by the last step of the *Transform* procedure we derive the empty program $T2$, 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 performed an experimental evaluation of our method on a benchmark set of programs acting on arrays, mostly taken from the literature [3, 12, 21, 27]. The results of our experiments, which are summarized in Tables 1 and 2, show that our approach is effective and quite efficient in practice.

Our verifier consists of a module, based on the C Intermediate Language (CIL) [32], which translates a C program together with the initial and error configurations, into a set of CLP(Array) facts, and a module for CLP(Array) program transformation that removes the interpreter and applies the *Transform* strategy. The latter module is implemented using the MAP system [29], 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 we have also indicated the properties we have verified).

Some programs deal with array initialization: program *init* initializes all the elements of the array to a constant, while *init-non-constant* and *init-sequence* use expressions which depend on the element position and on the preceding element, respectively. Program *init-partial* initializes only an initial portion of the array. Program *copy* performs the element-wise copy of an entire array to another array, while *copy-partial* copies only an initial portion of the array, and the program *copy-reverse* copies the array in reverse order. The program *max* computes the maximum of an array. The programs *sum* and *difference* perform the element-wise sum and difference, respectively, of two input arrays. The program *find* looks for a particular value inside an array and returns the position of its first occurrence, if any, or a negative value otherwise. The programs *find-first-non-null* and *first-not-null* are two programs which return the position of the first non-zero element. For these programs, differently from [12, 21], we prove that when the search succeeds, the returned position contains a non-zero element and we also proved that all the preceding elements are zero elements. The program *partition* copies non-negative and negative elements of the array into two distinct arrays. The programs *insertionsort-inner*, *bubblesort-inner*, and *selectionsort-inner* are based on textbook implementations of sorting algorithms. The source code of all the verification problems we have considered is available at `http://map.uniroma2.it/smc/`.

For verifying the above programs we have applied the *Transform* strategy using different generalization operators, which are based on the widening and convex hull operators. In particular the Gen_W and Gen_S operators use the *Widen* and *CHWidenSum* operators between constraints [14].

Program	*Code*	*Verified Property*
init	`for(i=0; i<n; i++)` `  a[i]=c;`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow a[i] = c$
init-partial	`for(i=0; i<k; i++)` `  a[i]=0;`	$\forall i.\ (0 \le i \wedge i < k \wedge k \le n)$ $\rightarrow a[i] = 0$
init-non-constant	`for(i=0; i<n; i++)` `  a[i]=2*i+c;`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow a[i] = 2*i+c$
init-sequence	`a[0]=7; i=1; while(i<n) {` `  a[i]=a[i-1]+1; i++;}`	$\forall i.\ (1 \le i \wedge i < n)$ $\rightarrow a[i] = a[i-1]+1$
copy	`for(i=0; i<n; i++)` `  a[i]=b[i];`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow a[i] = b[i]$
copy-partial	`for(i=0; i<k; i++)` `  a[i]=b[i];`	$\forall i.\ (0 \le i \wedge i < k \wedge k \le n)$ $\rightarrow a[i] = b[i]$
copy-reverse	`for(i=0; i<n; i++) b[i]=a[i];` `for(i=0; i<n; i++) a[i]=b[n-i-1];`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow a[i] = b[n-i-1]$
max	`m=a[0]; i=1; while(i<n) {` `  if(a[i]>m) m=a[i];  i++; }`	$\forall i.\ (0 \le i \wedge i < n \wedge n \ge 1)$ $\rightarrow m \ge a[i]$
sum	`for(i=0; i<n; i++)` `  c[i]=a[i]+b[i];`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow c[i] = a[i]+b[i]$
difference	`for(i=0; i<n; i++)` `  c[i]=a[i]-b[i];`	$\forall i.\ (0 \le i \wedge i < n)$ $\rightarrow c[i] = a[i]-b[i]$
find	`p=-1; for(i=0; i<n; i++)` `  if(a[i]==e) { p=i; break; }`	$(0 \le p \wedge p < n)$ $\rightarrow a[p] = e$
first-not-null	`s=n; for(i=0; i<n; ++i)` `  if(s==n && a[i]!=0) s=i;`	$(0 \le s \wedge s < n) \rightarrow (\ a[s] \neq 0 \wedge$ $(\forall i.\ (0 \le i \wedge i < s) \rightarrow a[i] = 0)\ )$
find-first-non-null	`p=-1; for(i=0; i<n; i++)` `  if(a[i]!=0) { p=i; break; }`	$(0 \le p \wedge p < n)$ $\rightarrow a[p] \neq 0$

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; }`
		$(\forall i.\ (0 \le i \wedge i < j)$
		$\rightarrow b[i] \ge 0) \quad \wedge$
		$(\forall i.\ (0 \le i \wedge i < k)$
		$\rightarrow c[i] < 0)$

insertionsort-inner	`x=a[i]; j=i-1;`
	`while(j>=0 && a[j]>x) {`
	`  a[j+1]=a[j]; --j; }`
		$\forall k.\ (0 \le i \wedge i < n \wedge j+1 < k \wedge k \le i)$
		$\rightarrow a[k] > x$

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; } }`
		$\forall k.\ (0 \le i \wedge i < n \wedge$
		$0 \le k \wedge k < j \wedge j = n-i-1)$
		$\rightarrow a[k] \le a[j]$

selectionsort-inner	`for(j=i+1; j<n; j++) {`
	`  if(a[i]>a[j]) { tmp=a[i];`
	`  a[i]=a[j]; a[j]=tmp; } }`
		$\forall k.(0 \le i \wedge i \le k \wedge k < n)$
		$\rightarrow a[k] \ge a[i]$

Table 1. Benchmark array programs. Variables `a,b,c` are arrays of integers of size `n`.

We have also combined these operators with a delay mechanism which, before starting the actual generalization process, introduces a definition which is computed by using convex hull alone, without widening. We denote by Gen_{WD} and Gen_{SD} the operators obtained by combining delayed generalization with the *Widen* and *CHWidenSum* operators, respectively.

In Table 2 we report the results obtained by applying *Transform* with the four generalization operators mentioned above. The first column contains references to papers where the program verification example has been considered.

The last four columns are labeled with the name of the generalization operator. For each program proved correct we report the time in seconds taken to verify the property of interest. By *unknown* we indicate that *Transform* derives a CLP(Array) program containing constrained facts different from '`incorrect`', and hence the satisfiability (or the unsatisfiability) of the corresponding verification conditions cannot be checked.

Program	*References*	Gen_W	Gen_{WD}	Gen_S	Gen_{SD}
init	[3, 12, 37]	*unknown*	0.06	0.10	0.08
init-partial	[3, 12]	*unknown*	0.06	0.07	0.08
init-non-constant	[3, 12, 27, 37]	*unknown*	0.06	0.22	0.22
init-sequence	[21, 27]	*unknown*	0.80	*unknown*	1.20
copy	[3, 12, 21, 27, 37]	*unknown*	0.27	0.33	0.29
copy-partial	[3, 12]	*unknown*	0.29	0.34	0.34
copy-reverse	[3, 12]	*unknown*	0.27	0.46	0.45
max	[21, 27]	*unknown*	0.31	0.24	0.33
sum		*unknown*	0.68	1.14	1.12
difference	[3]	*unknown*	0.66	1.15	1.11
find	[3, 12]	0.25	0.43	0.46	0.45
first-not-null	[21]	0.38	0.41	0.42	0.42
find-first-non-null	[3, 12]	1.24	1.87	1.94	1.93
partition	[12, 27, 37]	0.06	0.11	0.14	0.12
insertionsort-inner	[21, 27, 37]	0.21	0.26	0.45	0.43
bubblesort-inner		2.46	2.71	2.45	2.75
selectionsort-inner	[37]	7.20	6.40	7.23	7.16
	precision	7	17	16	17
	total time	11.80	15.65	17.14	18.48
	average time	1.69	0.92	1.07	1.09

Table 2. Verification results using the MAP system 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 run the whole benchmark and the *average time* per successful answer, respectively.

All 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 by using the Gen_W operator, which applies the widening operator alone, our method is only able to prove 7 programs out of 17. However, precision can be recovered by applying the convex hull operator when introducing new definitions, possibly combined with widening.

The best trade-off between precision and performance is provided by the Gen_{WD} operator which is able to prove all 17 programs with an average time of $0.92\,s$. In this case the use of the delay mechanism, which uses convex hull, suffices to compensate the weakness demonstrated by the use of widening alone. Note also that one program, *init-sequence*, can only be proved by applying operators which use delayed generalization. This confirms 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 determining (in our set of examples) only a slight increase of verification times.

A detailed comparison of the performance of our system with respect to the other verification systems referred to in Table 1 is difficult to make at this time because the 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 [8], where programs manipulating arrays were not considered. Some examples suggesting how arrays and recursively defined properties can be dealt with in our transformational approach were presented in [9], where, however, no automatic strategy was presented. In this paper we have shown that by applying a quite simple and general automated transformation strategy it is possible to prove most of the examples found in the literature, with reasonable performance. We are currently extending our strategy to deal with recursive programs, such as *quicksort*.

The idea of encoding imperative programs into CLP programs for reasoning about their properties was presented in various papers [15, 23, 34], which show that through CLP programs one can express in a simple manner both (i) the symbolic executions of imperative programs, and (ii) the invariants that hold during their executions. The peculiarity of our work with respect to [15, 23, 34] is that we use CLP program transformations to prove properties, instead of (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.

Now we briefly report on related papers which use abstract interpretations for finding invariants of programs that manipulate arrays. In [21], which builds upon [18], invariants are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. A similar approach is followed in [6] where a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs is introduced. In [16, 26] a predicate abstraction for inferring universally quantified properties of array el-

ements is presented, and in [20] the authors present a similar technique which uses template-based quantified abstract domains.

Methods based on abstract interpretation construct overapproximations, that is, invariants implied by the program executions. This approach has the advantage of being quite efficient because it fixes in advance a finite set of assertions where the invariants are searched for, but for the same reason it may lack flexibility as the abstraction should be re-designed when the verification fails.

Also theorem provers have been used for discovering invariants in programs which manipulate arrays and prove verification conditions generated from the programs. In particular, in [4] a satisfiability decision procedure for a decidable fragment of a theory of arrays is presented. That fragment is expressive enough to prove properties such as sortedness of arrays. In [24, 25, 31] the authors present some techniques based on theorem proving which may generate array invariants. In [37] a backward reachability analysis based on predicate abstraction and abstraction refinement is used for verifying assertions which are universally quantified over array indexes. Finally, we would like to mention that techniques based on Satisfiability Modulo Theory (SMT) have been applied for generating and verifying universally quantified properties over array variables (see, for instance, [1, 27]).

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, but the suitable assertions needed by the proof are generated on the fly.

Although the approach based on CLP program transformation shares many ideas and techniques with abstract interpretation and automated theorem proving, we believe that it has some distinctive features that make it quite appealing. Indeed, this paper and previous work (such as [8, 14, 34]) show that one can construct a framework where the generation of verification conditions and their verification can both be viewed as program transformations. The approach is 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 specialization. Moreover, optimizing transformations can be applied to improve the efficiency of verification. Finally, transformations can easily be composed together to derive very sophisticated verification techniques. For instance, in [8] it is shown that the *iteration* of specialization combined with the reversal of the direction used for constraint propagation can significantly improve the precision of verification.

In order to further validate our approach, we plan 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 which 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 transformations that guarantee equisatisfiability of verification conditions (like the

ones considered in this paper) together with other techniques for checking the satisfiability of constrained Horn clauses.

Acknowledgements

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

References

1. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *CAV '12*, LNCS 7358.
2. N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *SMT '12*, pages 3–11, 2012.
3. N. Bjørner, K. McMillan, and A. Rybalchenko. On solving universally quantified Horn clauses. In *SAS '13*, LNCS 7395, pages 105–125.
4. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI '06*, LNCS 3855, pages 427–442.
5. 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.
6. 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.
7. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*, pages 84–96. ACM, 1978.
8. E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *PEPM '13*, pages 43–52. ACM, 2013
9. 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.
10. G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, editor, *TACAS '99*, LNCS 1579, pages 223–239.
11. L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS '08*, LNCS 4963, pages 337–340.
12. I. Dillig, T. Dillig, and A. Aiken. Fluid updates: beyond strong vs. weak updates. In *ESOP'10*, 2010.
13. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
14. 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.
15. C. Flanagan. Automatic software model checking via constraint logic. In *Sci. Comput. Program.*, 50(1–3):253–270, 2004.
16. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL '02*, pages 191–202, New York, NY, USA, 2002. ACM.
17. 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.

18. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL '05*, pages 338–350. ACM, 2005.
19. 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.
20. 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.
21. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI '08*, pages 339–348, 2008.
22. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
23. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP '09*, LNCS 5732, pages 454–469. Springer, 2009.
24. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV '07*, LNCS 4590, pages 193–206, 2007.
25. 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.
26. S. K. Lahiri and R. E. Bryant. Predicate abstraction with indexed predicates. *ACM Trans. Comput. Log.*, 9(1), 2007.
27. D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *VMCAI 2013*, LNCS 7737.
28. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second Edition.
29. The MAP transformation system. http://www.iasi.cnr.it/∼proietti/system.html.
30. J. McCarthy. Towards a mathematical science of computation. *Information Processing : Proc. of IFIP 1962*, pages 21–28, Amsterdam, 1963. North Holland.
31. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS '08*, LNCS 4963, pages 413–427, 2008.
32. 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.
33. 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.
34. 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.
35. A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In *PADL '07*, LNCS 4354, pages 245–259.
36. C. J. Reynolds. *Theories of Programming Languages.* Cambridge Univ. Press 1998.
37. M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS '09*, LNCS 5673, pages 3–18. Springer, 2009.
38. M. H. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.