Verification of Imperative Programs through Transformation of Constraint Logic Programs

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

1 Introduction

In the last decade formal techniques have received a renewed attention as the basis of a methodology for increasing the reliability of software artifacts and reducing the cost of software production. In particular, great efforts have been made to devise automatic techniques such as *software model checking* [20], for verifying the correctness of programs with respect to their specifications.

In many software model checking techniques, the use of *constraints* has been very effective both for constructing models of programs and for reasoning about them [1, 7, 8, 10, 15, 17, 19, 30, 31]. Several kinds of constraints have been considered, such as equalities and inequalities over booleans, integers, reals, and finite or infinite trees. By using constraints we can represent in a symbolic, compact way the (possibly infinite) sets of values computed by programs and, in general, the sets of states which are reached during program executions. Then, by using powerful solvers specifically designed for the classes of constraints we have mentioned above, we can reason about program properties in an efficient way.

In this paper we consider a simple imperative programming language with integer and array variables and we use Constraint Logic Programming (CLP) [18] as a metalanguage for representing imperative programs, their executions, and the properties to be verified. We use constraints consisting of linear equalities and inequalities over integers. Note, however, that the method presented here is parametric with respect to the constraint domain which is used. By following an approach originally presented in [30], a given imperative program *prog* and its interpreter are first encoded as a CLP program. Then, the proofs of the properties of interest about the program *prog* are sought by analyzing that derived CLP program. In order to improve the efficiency of that analysis, it is advisable to first *compile-away* the CLP interpreter of the language in which *prog* is written. This is done by specializing the interpreter with respect to the given program *prog* using well-known *program specialization* techniques [21, 30].

In previous papers [8, 13] we have shown that program specialization can be used not only as a preprocessing step to improve the efficiency of program analysis, but also as a means of analysis on its own. In this paper, we extend that approach and we propose a verification method based on more general *unfold/fold transformation rules* for CLP programs [4, 11, 34].

Transformation-based verification techniques are very appealing because they are parametric with respect to both the programming languages in which programs are written, and the logics in which the properties of interest are specified. Moreover, since the output of a transformation-based verification method is a program which is *equivalent* to the given program with respect to the properties of interest, we can apply a *sequence* of transformations, thereby refining the analysis to the desired degree of precision (see, for instance, [8]).

 A. Lisitsa, A. Nemytykh (eds.), VPT 2013 (EPiC Series, vol. 16), pp. 30–41

The specific contributions of this paper are the following. We present a verification method based on a set of transformation rules which includes the rules for performing *conjunctive definition*, *conjunctive folding*, and *goal replacement*, besides the usual rules for *unfolding* and *constraint manipulation* which are used during program specialization. The rules for conjunctive definition and conjunctive folding allow us to introduce and transform new predicates defined in terms of *conjunctions* of old predicates, while program specialization can only deal with new predicates that correspond to specialized versions of exactly *one* old predicate. The goal replacement rule allows us to replace conjunctions of predicates and constraints by applying equivalences that hold in the least model of the CLP program at hand, while program specialization can only replace conjunctions of constraints.

By using these more powerful definition and folding rules, we extend the specialization-based verification method in the following two directions: (i) we verify programs with respect to specifications given by sets of CLP clauses (for instance, recursively defined relations among program variables), whereas program specialization can only deal with specifications given by constraints, and (ii) we verify programs manipulating arrays and other data structures by applying equivalences between predicates that axiomatize suitable properties of those data structures (for instance, the ones deriving from the axiomatization of the theory of arrays [28]).

The paper is organized as follows. In Section 2 we present our transformation-based verification method. First, we introduce a simple imperative language and we describe how correctness properties of imperative programs can be translated into predicates defined by CLP programs. We also present a general strategy for applying the transformation rules to CLP programs, with the objective of verifying the properties of interest. Next, we present two examples of application of our verification method. In particular, in Section 3 we show how we deal with specifications given by recursive CLP clauses, and in Section 4 we show how we deal with programs which manipulate arrays. Finally, in Section 5 we discuss the related work which has been recently done in the area of automatic program verification.

2 The Transformation-Based Verification Method

We consider an imperative C-like programming language with integer and array variables, assignments (`=`), sequential compositions (`;`), conditionals (`if` and `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* function from the set $\{0, \ldots, \mathit{dim}(a)-1\}$, where $\mathit{dim}(a)$ is the dimension of the array a, to the set of the integer numbers. The definition of the relation $\Longrightarrow$ is similar to the 'small step' operational semantics given in [32], 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 $\{\!\{\varphi_{init}\}\!\}\ \mathit{prog}\ \{\!\{\varphi_{error}\}\!\}$, where φ_{init} and φ_{error} are encoded by CLP predicates defined by (possibly recursive) clauses. We say that a program *prog* is *incorrect* with respect to φ_{init} and φ_{error}, whose free variables are assumed to be among $z_1, \ldots, z_r$, 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) $\langle\!\langle \ell_0 : c_0,\ \delta_{init} \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \mathtt{halt},\ \delta_h \rangle\!\rangle$, 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 this notion of correctness is equivalent to the usual notion of *partial correctness* specified by the Hoare triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$.

Our verification method is based on the formalization of the notion of program incorrectness by using a predicate `incorrect` defined by a CLP program.

In this paper a CLP program is a finite set of clauses of the form `A :- c,B`, where `A` is an atom, `c` is a constraint (that is, a possibly empty conjunction of linear equalities and inequalities over the integers), and `B` is a goal (that is, a possibly empty conjunction of atoms). The conjunction `c,B` is called a *constrained goal*. A clause of the form: `A :- c` is called a *constrained fact*. We refer to [18] for other notions of CLP with which the reader might be not familiar.

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 predicate `incorrect` is a consequence of the CLP program T defined by the following clauses:

```
incorrect :- initConf(X), reach(X).
reach(X) :- tr(X,X1), reach(X1).
reach(X) :- errorConf(X).
```

together with the clauses for the predicates `initConf(X)`, `errorConf(X)`, and `tr(X,X1)`. They 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,X1)` encodes the transition relation $\Longrightarrow$. (Note that in order to define `initConf(X)`, `errorConf(X)`, and `tr(X,X1)` and, in particular, to represent operations over the integer variables and the elements of arrays, we need constraints.) The predicate `reach(X)` holds if an error configuration `Y` such that `errorConf(Y)` holds, can be reached from the configuration `X`.

The imperative program *prog* is correct with respect to the properties φ_{init} and φ_{error} iff `incorrect` $\notin M(T)$, where $M(T)$ denotes the *least model* of program T [18]. Due to the presence of integer variables and array variables, $M(T)$ is in general an infinite model, and both the bottom-up and top-down evaluation of the query `incorrect` may not terminate. In order to deal with this difficulty, we propose an approach to program verification which is symbolic and, by using program transformations, allows us to avoid the exhaustive exploration of the possibly infinite space of reachable configurations.

Our verification method consists in applying to program T a sequence of program transformations that preserve the least model $M(T)$ [11]. In particular, we apply the following *transformation rules*, collectively called *unfold/fold rules*: (i) (*conjunctive*) *definition*, (ii) *unfolding*, (iii) *goal replacement*, (iv) *clause removal*, and (v) (*conjunctive*) *folding*. Our verification method is made out of the following two steps.

Step (A): *Removal of the Interpreter.* Program T is *specialized* with respect to the given *prog* (on which `tr` depends), `initConf`, and `errorConf`, thereby deriving a new program $T1$ such that: (i) `incorrect` $\in M(T)$ iff `incorrect` $\in M(T1)$, and (ii) `tr` does not occur explicitly in $T1$ (in this sense we say that the interpreter is removed or compiled-away).

Step (B): *Propagation of the Initial and Error Properties.* By applying a sequence of unfold/fold transformation rules, the CLP program $T1$ is transformed into a new CLP program $T2$ such that `incorrect` holds in $M(T2)$ iff *prog* is incorrect with respect to the given initial and error properties. The objective of Step (B) is to propagate the initial and the error properties so as to derive a program $T2$ where the predicate `incorrect` is defined by either (i) the fact '`incorrect.`' (in which case *prog* is incorrect), or (ii) the empty set of clauses (in which case *prog* is correct). In the case where neither (i) nor (ii) holds, that is, in program $T2$ the predicate `incorrect` is defined by a non-empty set of clauses not containing the fact '`incorrect.`', we cannot conclude anything about the correctness of *prog* and, similarly to what has been proposed in [8], we iterate Step (B) 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.

Steps (A) and (B) are both instances of the *Transform* strategy outlined in Figure 1 below.

Input: Program P.
Output: Program *TransfP* such that $\texttt{incorrect} \in M(P)$ iff $\texttt{incorrect} \in M(\textit{TransfP})$.

INITIALIZATION:
$\textit{TransfP} := \emptyset$; $\textit{InDefs} := \{\texttt{incorrect:- c, G}\}$; $\textit{Defs} := \textit{InDefs}$;

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

UNFOLDING: Apply the unfolding rule at least once, and derive from C a set $U(C)$ of clauses;

GOAL REPLACEMENT: Apply a sequence of goal replacements, 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 *NewDefs* 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;

$\textit{InDefs} := \textit{InDefs} - \{C\}$; $\textit{TransfP} := \textit{TransfP} \cup F(C)$;

end-while;

REMOVAL OF USELESS CLAUSES:
Remove from *TransfP* all clauses whose head predicate is useless.

Figure 1: The *Transform* strategy.

In particular, the application of the *Transform* strategy for performing Step (A) coincides with the fully automatic specialization strategy presented in [8]. In the *Transform* strategy we make use of the following rules, where P is the input CLP program, and *Defs* is a set of clauses, called *definition clauses*, constructed as we indicate in that strategy.

Definition Rule. By this rule we introduce a clause of the form `newp(X) :- c,G`, where `newp` is a new predicate symbol, `X` is a tuple of variables occurring in `(c,G)`, `c` is a constraint, and `G` is a non-empty conjunction of atoms.

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

Goal Replacement Rule. If a constrained goal $\texttt{c}_1, \texttt{G}_1$ occurs in the body of a clause C, and $M(P) \models \forall\,(\texttt{c}_1, \texttt{G}_1 \leftrightarrow \texttt{c}_2, \texttt{G}_2)$, then we derive a new clause D by replacing $\texttt{c}_1, \texttt{G}_1$ by $\texttt{c}_2, \texttt{G}_2$ in the body of C.

The equivalences which are needed for goal replacements are called *laws* and their validity in $M(P)$ can be proved once and for all, before applying the *Transform* strategy.

Folding Rule. Given a clause E of the form: `H :- e, L, Q, R` and a clause D in *Defs* of the form `K :- d, D` such that: (i) for some substitution ϑ, $\mathtt{Q} = \mathtt{D}\,\vartheta$, and (ii) $\forall\,(\mathtt{e} \rightarrow \mathtt{d}\,\vartheta)$ holds, then by folding E using D we derive $\mathtt{H :- e, L, K}\,\vartheta\mathtt{, R}$.

Removal of Useless Clauses. The set of *useless predicates* in a given program Q is the greatest set U of predicates occurring in Q such that `p` is in U iff every clause with head predicate `p` is of the form $\mathtt{p(X)\ :\text{-}\ c, G_1, q(Y), G_2}$, for some `q` in U. A clause in a program Q is *useless* if the predicate of its head is useless in Q.

The termination of the *Transform* strategy is guaranteed by suitable techniques for controlling the unfolding and the introduction of new predicates. We refer to [25] for a survey of techniques which ensure the finiteness of unfolding. The introduction of new predicates is controlled by applying *generalization operators* based on various notions, such as *widening*, *convex hull*, *most specific generalization*, and *well-quasi ordering*, which have been proposed for analyzing and transforming CLP programs (see, for instance, [7, 9, 14, 29]).

The correctness of the strategy with respect to the least model semantics directly follows from the fact that the application of the transformation rules complies with some suitable conditions that guarantee the preservation of that model [11].

Theorem 1. (Termination and Correctness of the *Transform* strategy) (i) *The Transform strategy terminates.* (ii) *Let program TransfP be the output of the Transform strategy applied on the input program P. Then,* $\mathtt{incorrect} \in M(P)$ *iff* $\mathtt{incorrect} \in M(\mathit{TransfP})$.

3 Verification of Recursively Defined Properties

In this section we will show, through an example, that our verification method can be used when the initial properties and the error properties are specified by (possibly recursive) CLP clauses, rather than by constraints only (as done, for instance, in [8]). In order to deal with that kind of properties, during the DEFINITION & FOLDING phase of the *Transform* strategy, we allow ourselves to introduce new predicates which are defined by clauses of the form: `Newp :- c, G`, where `Newp` is an atom with a new predicate symbol, `c` is a constraint, and `G` is a conjunction of *one or more* atoms. This kind of predicate definitions allows us to perform program verifications that cannot be done by the technique presented in [8], where the goal `G` is assumed to be a single atom.

Let us consider the following program *GCD* that computes the greatest common divisor z of two positive integers m and n, denoted $gcd(m, n, z)$.

GCD: ℓ_0: $x = m\,;$
ℓ_1: $y = n\,;$
ℓ_2: `while` $(x \neq y)$ $\{$ `if` $(x > y)$ $x{=}x{-}y\,;$ `else` $y{=}y{-}x\,;\}\,;$
ℓ_3: $z = x\,;$
ℓ_h: `halt`

We also consider the incorrectness triple $\{\!\{\varphi_{init}(m,n)\}\!\}\ GCD\ \{\!\{\varphi_{error}(m,n,z)\}\!\}$, where: (i) $\varphi_{init}(m,n)$ is $m \geq 1 \wedge n \geq 1$, and (ii) $\varphi_{error}(m,n,z)$ is $\exists\, d\,(gcd(m,n,d) \wedge d \neq z)$. These properties φ_{init} and φ_{error} are defined by the following CLP clauses 1 and 2–5, respectively:

1. `phiInit(M,N) :- M≥1, N≥1.`
2. `phiError(M,N,Z) :- gcd(M,N,D), D≠Z.`
3. `gcd(X,Y,D) :- X>Y, X1=X−Y, gcd(X1,Y,D).`
4. `gcd(X,Y,D) :- X<Y, Y1=Y−X, gcd(X,Y1,D).`
5. `gcd(X,Y,D) :- X=Y, Y=D.`

The predicates `initConf` and `errorConf` specifying the initial and the error configurations, respectively, are defined by the following clauses:

6. `initConf(cf(cmd(0, asgn(int(x), int(m))),`
   `[[int(m), M], [int(n), N], [int(x), X], [int(y), Y], [int(z), Z]])) :- phiInit(M, N).`
7. `errorConf(cf(cmd(h, halt),`
   `[[int(m), M], [int(n), N], [int(x), X], [int(y), Y], [int(z), Z]])) :- phiError(M, N, Z).`

Thus, the CLP program encoding the given incorrectness triple consists of clauses 1–7 above, together with the clauses defining the predicates `incorrect`, `reach`, and `tr`.

Now we perform Step (A) of our verification method, which consists in the removal of the interpreter, and we derive the following CLP program:

8. `incorrect :- M≥1, N≥1, X=M, Y=N, new1(M, N, X, Y, Z).`
9. `new1(M, N, X, Y, Z) :- X>Y, X1=X−Y, new1(M, N, X1, Y, Z).`
10. `new1(M, N, X, Y, Z) :- X<Y, Y1=Y−X, new1(M, N, X, Y1, Z).`
11. `new1(M, N, X, Y, Z) :- X=Y, Z=X, Z≠D, gcd(M, N, D).`

By moving the constrained atom '`Z≠D, gcd(M, N, D)`' from the body of clause 11 to the body of clause 8, we can rewrite clauses 8 and 11 as follows (this rewriting is correct because in clauses 9 and 10 the predicate `new1` modifies neither the value of `M` nor the value of `N`):

8r. `incorrect :- M≥1, N≥1, X=M, Y=N, Z≠D, gcd(M, N, D), new1(M, N, X, Y, Z).`
11r. `new1(M, N, X, Y, Z) :- X=Y, Z=X.`

Note that we could avoid performing the above rewriting and obtain a similar program where the constraints characterizing the initial and the error properties occur in the same clause by starting our derivation from a more general definition of the reachability relation. However, an in-depth analysis of this variant of our verification method is beyond the scope of this paper.

Now we will perform Step (B) of the verification method by applying the *Transform* strategy to the derived program consisting of clauses $\{3, 4, 5, 8r, 9, 10, 11r\}$. Initially, we have that the sets *InDefs* and *Defs* of definition clauses are both equal to $\{8r\}$.

Unfolding. We start off by unfolding clause 8r w.r.t. the atom `new1(M, N, X, Y, Z)`, and we get:

12. `incorrect :- M≥1, N≥1, X=M, Y=N, X>Y, X1=X−Y, Z≠D, gcd(M, N, D), new1(M, N, X1, Y, Z).`
13. `incorrect :- M≥1, N≥1, X=M, Y=N, X<Y, Y1=Y−X, Z≠D, gcd(M, N, D), new1(M, N, X, Y1, Z).`
14. `incorrect :- M≥1, N≥1, X=M, Y=N, X=Y, Z=X, Z≠D, gcd(M, N, D).`

By unfolding clauses 12, 13, and 14 w.r.t. the atom `gcd(M, N, D)`, we derive:

15. `incorrect :- M≥1, N≥1, M>N, X1=M−N, Z≠D, gcd(X1, N, D), new1(M, N, X1, N, Z).`
16. `incorrect :- M≥1, N≥1, M<N, Y1=N−M, Z≠D, gcd(M, Y1, D), new1(M, N, M, Y1, Z).`

(The unfolding of clause 14 produces the empty set of clauses because the constraint '`X=M, Z=X, Z≠D, M=D`' is unsatisfiable.) The Goal Replacement and Clause Removal phases leave the set of clauses produced by the Unfolding phase unchanged, because no laws are available for the predicate `gcd`.

Definitions & Folding. In order to fold clauses 15 and 16, we perform a generalization step and we introduce a new predicate defined by the following clause:

17. `new2(M, N, X, Y, Z, D) :- M≥1, N≥1, Z≠D, gcd(X, Y, D), new1(M, N, X, Y, Z).`

The body of this clause 17 is the most specific generalization of the bodies of clause 8r (which is the only clause in *Defs*), and clauses 15 and 16 (which are the clauses to be folded). Now, clauses 15 and 16 can be folded by using clause 17, thereby deriving:

18. `incorrect :- M≥1, N≥1, M>N, X1=M−N, Z≠D, new2(M, N, X1, N, Z, D).`
19. `incorrect :- M≥1, N≥1, M<N, Y1=N−M, Z≠D, new2(M, N, M, Y1, Z, D).`

Clause 17 defining the new predicate `new2` is added to *Defs* and *InDefs* and, since the latter set is not empty, we perform a new iteration of the while-loop body of the *Transform* strategy. UNFOLDING. By unfolding clause 17 w.r.t. `new1(M,N,X,Y,Z)` and then unfolding the resulting clauses w.r.t. `gcd(X,Y,Z)`, we derive:

20. `new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X>Y, X1=X−Y, Z≠D, gcd(X1,Y,D), new1(M,N,X1,Y,Z).`
21. `new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X<Y, Y1=Y−X, Z≠D, gcd(X,Y1,D), new1(M,N,X,Y1,Z).`

DEFINITION & FOLDING. Clauses 20 and 21 can be folded by using clause 17, and we derive:

22. `new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X>Y, X1=X−Y, Z≠D, new2(M,N,X1,Y,Z).`
23. `new2(M,N,X,Y,Z,D) :- M≥1, N≥1, X<Y, Y1=Y−X, Z≠D, new2(M,N,X,Y1,Z).`

No new predicate definition is introduced, and the *Transform* strategy exits the while-loop. The final program *TransfP* is the set $\{18, 19, 22, 23\}$ of clauses, which contains no constrained facts. Hence both predicates `incorrect` and `new2` are useless and all clauses of *TransfP* can be removed. Thus, the *Transform* strategy terminates with $\mathit{TransfP}=\emptyset$ and we conclude that the imperative program *GCD* is correct w.r.t. the given initial and error properties.

4 Verification of Array Programs

In this section we apply our verification method to the following program *ArrayMax* which computes the maximal element of an array:

ArrayMax: ℓ_0: `while` $(i<n)$ { `if` $(a[i] > max)$ $max = a[i]$;
$i = i{+}1$; };
ℓ_h: `halt`

We consider the following incorrectness triple: $\{\!\{\varphi_{init}(i,n,a,max)\}\!\}$ *ArrayMax* $\{\!\{\varphi_{error}(n,a,max)\}\!\}$ where: (i) $\varphi_{init}(i,n,a,max)$ is $i=0 \wedge n=dim(a) \wedge n\geq 1 \wedge max=a[i]$, and (ii) $\varphi_{error}(n,a,max)$ is $\exists k\ (0\leq k<n \wedge a[k]>max)$.

First, we construct a CLP program T which encodes the above incorrectness triple, similarly to what has been done in Section 3. The predicates `initConf(X)` and `errorConf(X)` specifying the initial and the error configurations, respectively, are defined by the following clauses:

1. `initConf(cf(cmd(0,asgn(int(x),int(0))),`
   `[[int(i),I],[int(n),N],[array(a),(A,N)],[int(max),Max]])) :- phiInit(I,N,A,Max).`
2. `errorConf(cf(cmd(h,halt),`
   `[[int(i),I],[int(n),N],[array(a),(A,N)],[int(max),Max]])) :- phiError(N,A,Max).`
3. `phiInit(I,N,A,Max) :- I=0, N≥1, read((A,N),I,Max).`
4. `phiError(N,A,Max) :- K≥0, N>K, Z>Max, read((A,N),K,Z).`

Now we start off by applying 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 :- I=0, N≥1, read((A,N),I,Max), new1(I,N,A,Max).`
6. `new1(I,N,A,Max) :- I1=I+1, I<N, I≥0, M>Max, read((A,N),I,M), new1(I1,N,A,M).`
7. `new1(I,N,A,Max) :- I1=I+1, I<N, I≥0, M≤Max, read((A,N),I,M), new1(I1,N,A,Max).`
8. `new1(I,N,A,Max) :- I≥N, K≥0, N>K, Z>Max, read((A,N),K,Z).`

As indicated in [8], in order to propagate the error property, we 'reverse' the derived program $T1$ and we get the following program $T1_{rev}$:

rev1. `incorrect :- b(U), r2(U).`
rev2. `r2(V) :- trans(U,V), r2(U).`
rev3. `r2(U) :- a(U).`

where the predicates `a`, `b`, and `trans` are defined as follows:

s4. `a([new1, I, N, A, Max]) :- I=0, N≥1, read((A, N), I, Max)`
s5. `trans([new1, I, N, A, Max], [new1, I1, N, A, M]) :-`
`I1=I+1, I<N, I≥0, M>Max, read((A, N), I, M).`
s6. `trans([new1, I, N, A, Max], [new1, I1, N, A, Max]) :-`
`I1=I+1, I<N, I≥0, M≤Max, read((A, N), I, M).`
s7. `b([new1, I, N, A, Max]) :- I≥N, K≥0, K<N, Z>Max, read((A, N), K, Z).`

The transformation from $T1$ to $T1_{rev}$ is correct in the sense that `incorrect` $\in M(T1)$ iff `incorrect` $\in M(T1_{rev})$. This equivalence holds because: (i) in program $T1$ the predicate `incorrect` is defined in terms of the predicate `new1` that encodes the reachability relation from an error configuration to an initial configuration, and (ii) in program $T1_{rev}$ the predicate `incorrect` is defined in terms of the predicate `r2` that also encodes the reachability relation, but this time the encoding is 'in the reversed direction', that is, from an initial configuration to an error configuration.

Now let us apply Step (B) of our verification method starting from the program $T1_{rev}$.

UNFOLDING. First we unfold clause rev1 w.r.t. the atom `b(U)`, and we get:

9. `incorrect :- I≥N, K≥0, K<N, Z>Max, read((A, N), K, Z), r2([new1, I, N, A, Max]).`

Neither GOAL REPLACEMENT nor CLAUSE REMOVAL is applied.

DEFINITION & FOLDING. In order to fold clause 9 we introduce the following clause:

10. `new2(I, N, A, Max, K, Z) :- I≥N, K≥0, K<N, Z>Max, read((A,N), K, Z), r2([new1, I, N, A, Max]).`

By folding clause 9 using clause 10, we get:

11. `incorrect :- I≥N, K≥0, K<N, Z>Max, new2(I, N, A, Max, K, Z).`

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, clause 10 belongs to *InDefs*).

UNFOLDING. After some unfoldings from clause 10 we get the following clauses:

12. `new2(I1, N, A, M, K, Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
`read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

13. `new2(I1, N, A, Max, K, Z) :- I1=I+1, N=I1, K≥0, K<I1, M≤Max, Z>Max,`
`read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

GOAL REPLACEMENT. We use the following law which is a consequence of the fact that arrays are finite functions:

(GR) `read((A, N), K, Z), read((A, N), I, M)` $\leftrightarrow$
`(K=I, Z=M, read((A, N), K, Z))` $\vee$ `(K≠I, read((A, N), K, Z), read((A, N), I, M))`

Thus, (i) we replace the conjunction of atoms '`read((A, N), K, Z), read((A, N), I, M)`' occurring in the body of clause 12 by the right hand side of law (GR), and then (ii) we split the derived clause with disjunctive body into the following two clauses, each of which corresponds to a disjunct of the right hand side of (GR). We get the following clauses:

12.1 `new2(I1, N, A, M, K, Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
`K=I, M=Z, read((A, N), K, Z), r2([new1, I, N, A, Max]).`
12.2 `new2(I1, N, A, M, K, Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,`
`K≠I, read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

CLAUSE REMOVAL. The constraint '`Z>M, M=Z`' in the body of clause 12.1 is unsatisfiable. Hence, this clause is removed from *TranfP*. By simplifying the constraints in clause 12.2 we get:

14. `new2(I1, N, A, M, K, Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
`read((A, N), K, Z), read((A, N), I, M), r2([new1, I, N, A, Max]).`

By applying similar goal replacements and clause removals, from clause 13 we get:

15. `new2(I1,N,A,Max,K,Z) :- I1≤I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
`read((A,N),K,Z), read((A,N),I,M), r2([new1,I,N,A,Max]).`

DEFINITION & FOLD. In order to fold clause 14, we introduce the following definition:

16. `new3(I,N,A,Max,K,Z) :- K≥0, K<N, K<I, Z>Max, read((A,N),K,Z)), r2([new1,I,N,A,Max]).`

Clause 16 is obtained from clauses 10 and 14 by applying a generalization operator called *WidenSum* [14], which is a variant of the classical widening operator [5]. Clause 16 can be used also for folding clause 15, and by folding clauses 14 and 15 using clause 16, we get:

17. `new2(I1,N,A,Max,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M>Max, Z>M,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`

18. `new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I, M≤Max, Z>Max,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`

Now we perform the third iteration of the body of the while-loop of the strategy. After some unfolding, goal replacement, clause removal, and folding steps, from clause 16 we get:

19. `new3(I1,N,A,M,K,Z) :- I1=I+1, K≥0, K+1<I1, N≥I1, M>Max, Z>M,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`

20. `new3(I1,N,A,Max,K,Z) :- I1=I+1, K≥0, K+1<I1, N≥I1, M≤Max, Z>Max,`
`read((A,N),I,M), new3(I,N,A,Max,K,Z).`

Since we did not introduce any new definition, and no clause remains to be processed (indeed, the set *InDefs* of definitions is empty), the *Transform* strategy exits the while-loop and we get the program consisting of the set $\{11, 17, 18, 19, 20\}$ of clauses.

Since no clause in this set is a constrained fact, by the final phase of removing the useless clauses we get a final program consisting of the empty set of clauses. Thus, the program *ArrayMax* is correct with respect to the given φ_{init} and φ_{error} properties.

5 Related Work and Conclusions

The verification method presented in this paper is an extension of the one introduced in [8], where Constraint Logic Programming (CLP) and iterated specialization have been used to define a general verification framework that is parametric with respect to the programming language and the logic used for specifying the correctness properties. The main novelties of this paper are the following ones: (i) we have considered imperative programs acting on integer variables as well as array variables, and (ii) we have allowed a more expressive specification language, in which one can write properties about elements of arrays and, in general, elements of complex data structures.

In order to deal with this more general setting, we have defined the operational semantics of array manipulation, and we have also considered powerful transformation rules, such as conjunctive definition, conjunctive folding, and goal replacement. These transformation rules together with some strategies for guiding their application, have been implemented in the MAP transformation system [26], so that the proofs of program correctness have been performed in a semi-automatic way.

The use of constraint-based techniques for program verification is not novel. Indeed, CLP programs have been successfully applied to perform model checking of both finite and infinite state systems [10, 12, 14] because through CLP programs one can express in a simple manner both (i) the symbolic executions of imperative programs and (ii) the invariants which hold during their executions. Moreover, there are powerful CLP-based tools, such as ARMC [31], TRACER [19], and HSF [17], that can be used for performing model checking of imperative programs. These tools are fully automatic, but they are applicable to classes of programs and properties that are much more limited than those considered in this paper. We have shown in [8] that, by focusing on verification tasks similar to those considered by ARMC, TRACER, and HSF, we can design a fully automatic, transformation-based verification technique whose effectiveness is competitive to the one of the above mentioned tools.

Our rule-based program transformation technique is also related to *conjunctive partial deduction* (CPD) [9], a technique for the specialization of logic programs with respect to conjunctions of atoms. There are, however, some substantial differences between CPD and the approach we have presented here. First, CPD is not able to specialize logic programs with constraints and, thus, it cannot be used to prove the correctness of the *GCD* program where the role of constraints is crucial. Indeed, using the ECCE conjunctive partial deduction system [24] for specializing the program consisting of clauses $\{3, 4, 5, 8r, 9, 10, 11r\}$ with respect to the query `incorrect`, we obtain a residual program where the predicate `incorrect` is not useless. Thus, we cannot conclude that the atom `incorrect` does not belong to the least model of the program, and thus we cannot conclude that the program is correct. One more difference between CPD and our technique is that we may use goal replacement rules which allow us to evaluate terms over domain-specific theories. In particular, we can apply the goal replacement rules using well-developed theories for data structures like arrays, lists, heaps and sets (see [3, 27, 16, 2, 33, 36] for some formalizations of these theories).

An alternative, systemic approach to program transformation is supercompilation [35], which considers programs as machines. A supercompiler runs a program and, while it observes its behavior, produces an equivalent program without performing stepwise transformations of the original program.

The verification method presented in this paper is also related to several other methods for verifying properties of imperative programs acting on arrays. Those methods use techniques based on abstract interpretation, theorem proving and, in particular, Satisfiability Modulo Theory (see, for instance, [6, 22, 23]).

The application of the powerful transformation rules we have considered in this paper enables the verification of a larger class of properties, but it does not entirely fit into the automated strategy used in [8]. In the future we intend to consider the issue of designing fully mechanizable strategies for guiding the application of our program transformation rules. In particular, we want to study the problem of devising suitable unfolding strategies and generalization operators, by adapting the techniques already developed for program transformation. We also envisage that the application of the laws used by the goal replacement rule can be automated by importing in our framework the techniques used in the fields of Theorem Proving and Term Rewriting. For some specific theories we could also apply the goal replacement rule by exploiting the results obtained by external theorem provers or Satisfiability Modulo Theory solvers.

We also plan to address the issue of proving correctness of programs acting on *dynamic data structures* such as lists or heaps, looking for a set of suitable goal replacement laws which axiomatize those structures.

Acknowledgements

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

References

[1] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI '07*, Lecture Notes in Computer Science 4349, pages 378–394. Springer, 2007.

[2] R. S. Bird. An introduction to the theory of lists. In *Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design*, pages 5–42. Springer-Verlag New York, Inc., 1987.

[3] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI'06, Charleston, SC, USA, January 8-10, 2006*, Lecture Notes in Computer Science 3855. Springer, 2006.

[4] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings of the 4th ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.

[6] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th ACM Symposium on Principles of programming languages. POPL'11*, pages 105–118, 2011.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 84–96. ACM Press, 1978.

[8] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Programs via Iterated Specialization. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13*, pages 43–52, 2013.

[9] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens, and M. H. Sørensen. Conjunctive partial deduction: Foundations, control, algorithms, and experiments. *Journal of Logic Programming*, 41(2–3):231–277, 1999.

[10] G. Delzanno and A. Podelski. Model checking in CLP. In R. Cleaveland, ed., *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*, Lecture Notes in Computer Science 1579, pages 223–239. Springer-Verlag, 1999.

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

[12] F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In *Proceedings of the ACM SIGPLAN Workshop on Verification and Computational Logic VCL'01, Florence (Italy)*, Technical Report DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.

[13] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. In G. Delzanno and I. Potapov, eds., *Proceedings of the 5th International Workshop on Reachability Problems (RP 2011), September 28-30, 2011, Genova, Italy*, Lecture Notes in Computer Science 6945, pages 165–179. Springer, 2011.

[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. Special Issue on the 25th Annual GULP Conference*, 13(2):175–199, 2013.

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

[16] 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.

[17] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In *Proc. of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '12*, Lecture Notes in Computer

Science 7214, pages 549–551. Springer, 2012.

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

[19] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa. TRACER: A symbolic execution tool for verification. In *CAV*, Lecture Notes in Computer Science 5732, pages 758–766. Springer, 2012.

[20] R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

[21] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.

[22] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering. FASE 2009*, Lecture Notes in Computer Science 5503, pages 470–485. Springer, 2009.

[23] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2013, Rome, Italy, January 20-22, 2013.*, Lecture Notes in Computer Science 7737, pages 169–188. Springer, 2013.

[24] M. Leuschel. The ECCE partial deduction system and the DPPD library of benchmarks, Release 3, Nov. 2000. Available from `http://www.ecs.soton.ac.uk/~mal`.

[25] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4&5):461–515, 2002.

[26] MAP. 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`.

[27] J. McCarthy. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, pages 33–70. North-Holland, 1963.

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

[29] J. C. Peralta and J. P. Gallagher. Convex hull abstractions in specialization of CLP programs. In M. Leuschel, ed., *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR 2002, Madrid, Spain, September 17–20, 2002, Revised Selected Papers*, Lecture Notes in Computer Science 2664, pages 90–108, 2003.

[30] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, ed., *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998*, Lecture Notes in Computer Science 1503, pages 246–261. Springer, 1998.

[31] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In M. Hanus, ed., *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, pages 245–259. Springer, 2007.

[32] C. J. Reynolds. *Theories of Programming Languages.* Cambridge University Press, 1998.

[33] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS'02, pages 55–74. IEEE Computer Society, 2002.

[34] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, ed., *Proceedings of the Second International Conference on Logic Programming (ICLP'84)*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.

[35] V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3):292–325, 1986.

[36] M. Wirsing. Algebraic specification. In J. Van Leeuwen, ed., *Handbook of Theoretical Computer Science*, volume B, pages 675–788. Elsevier, 1990.