Verification of Imperative Programs by Constraint Logic Program Transformation

Emanuele De Angelis
DEC, University 'G. D'Annunzio', Pescara, Italy
emanuele.deangelis@unich.it

Fabio Fioravanti
DEC, University 'G. D'Annunzio', Pescara, Italy
fioravanti@unich.it

Alberto Pettorossi
DICII, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it

Maurizio Proietti
IASI-CNR, Rome, Italy
maurizio.proietti@iasi.cnr.it

We present a method for verifying partial correctness properties of imperative programs that manipulate integers and arrays by using techniques based on the transformation of constraint logic programs (CLP). We use CLP as a metalanguage for representing imperative programs, their executions, and their properties. First, we encode the correctness of an imperative program, say *prog*, as the negation of a predicate `incorrect` defined by a CLP program T. By construction, `incorrect` holds in the least model of T if and only if the execution of *prog* from an initial configuration eventually halts in an error configuration. Then, we apply to program T a sequence of transformations that preserve its least model semantics. These transformations are based on well-known transformation *rules*, such as *unfolding* and *folding*, guided by suitable transformation *strategies*, such as *specialization* and *generalization*. The objective of the transformations is to derive a new CLP program *TransfT* where the predicate `incorrect` is defined either by (i) the fact '`incorrect.`' (and in this case *prog* is *not* correct), or by (ii) the empty set of clauses (and in this case *prog* is correct). In the case where we derive a CLP program such that neither (i) nor (ii) holds, we iterate the transformation. Since the problem is undecidable, this process may not terminate. We show through examples that our method can be applied in a rather systematic way, and is amenable to automation by transferring to the field of program verification many techniques developed in the field of program transformation.

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 [40]. In particular, a massive effort has been made to devise automatic verification techniques, such as *software model checking* [31], for proving the correctness of programs with respect to their specifications.

In many software model checking techniques, the notion of a *constraint* has been shown to be very effective, both for constructing models of programs and for reasoning about them [2, 9, 12, 14, 20, 24, 30, 43, 44]. Several types of constraints have been considered, such as equalities and inequalities over the booleans, the integers, the reals, the finite or infinite trees. By using constraints we can represent in a symbolic, compact way (possibly infinite) sets of values computed by programs and, more in general, sets of states reached during program executions. Then, in order to reason about program properties in an efficient way, we can use solvers specifically designed for the various classes of constraints we have mentioned above.

In this paper we consider a simple imperative programming language with integer and array variables, and we adopt Constraint Logic Programming (CLP) [28] as a metalanguage for representing imperative

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 186–210, doi:10.4204/EPTCS.129.12

programs, their executions, and their properties. We use constraints consisting of equalities and inequalities over the integers, but the method presented here is parametric with respect to the constraint domain which is used. By following an approach first presented in [43], a given imperative program *prog* and its interpreter are encoded as a CLP program. Then, the proofs of the properties of interest about the program *prog* are sought by analyzing the derived CLP program. In order to improve the efficiency of 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* (also known as *partial evaluation*) techniques [33, 43].

We have shown in previous work [11, 12, 18] 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 methodology based on more general, semantics preserving *unfold/fold transformation rules* for CLP programs [6, 15, 50].

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 of a program is an *equivalent* 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.

Our approach can be summarized as follows. Suppose we are given: (i) an imperative program *prog*, (ii) a predicate `initConf` expressing the property, called the *initial property*, holding in the configuration from which the execution of *prog* starts, and (iii) a predicate `errorConf` expressing the property, called the *error property*, holding in the configuration which should *not* be reached at the end of the execution of *prog*. The partial correctness of *prog* is defined as the negation of a predicate `incorrect` specified by the following CLP program T:

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

where: (i) `reach(X)` holds iff an error configuration can be reached from the configuration `X`, and (ii) `tr(X,X1)` holds iff the configuration `X1` can be reached in one step from the configuration `X` via the transition relation that defines the operational semantics of the imperative language. Thus, `incorrect` holds iff there exists an error configuration that can be reached from an initial configuration.

The verification method we propose in this paper is made out of the following two steps.

Step (A). This step, called the *removal of the interpreter*, consists in specializing the program T (which includes the clauses defining the predicate `tr`) with respect to the given program *prog* and properties `initConf` and `errorConf`. After this specialization step we derive from program T a new program T_A, where there is no reference to the predicate `tr` (and in this sense we say that during this Step (A) the interpreter is removed or 'compiled-away').

Step (B). This step, called the *propagation of the initial and error properties*, consists in applying a sequence of unfold/fold transformation rules, and deriving from the CLP program T_A a new CLP program T_B such that `incorrect` holds in T_B if and only if *prog* is *not* correct with respect to the given initial and error properties. The objective of this Step (B) is to derive a program T_B where the predicate `incorrect` is defined by: either (i) the fact '`incorrect.`' (and in this case *prog* is not correct), or (ii) the empty set of clauses (and in this case *prog* is correct). If neither Case (i) nor Case (ii) holds, that is, in program T_B the predicate `incorrect` is defined by a non-empty set of clauses which does not contain the fact '`incorrect.`', we can conclude neither the correctness nor the incorrectness of *prog*. Thus, similarly to what has been proposed in [12], we continue our verification method by iterating this Step (B) in the

hope of deriving a program where either Case (i) or Case (ii) holds. Obviously, due to undecidability limitations, it may be the case that we never derive such a program.

During Step (B) we apply transformation rules that are more powerful than those needed for program specialization and, in particular, for the removal of the interpreter done during Step (A). Indeed, the rules used during Step (B) include the *conjunctive definition* and the *conjunctive folding* rules and they allow us to introduce and transform new predicate definitions that correspond to *conjunctions* of old predicates, while program specialization can deal only with new predicates that correspond to specialized versions of *one* old predicate. During Step (B) we use also the *goal replacement* rule, which allows us to replace conjunctions of constraints and predicates by applying equivalences that hold in the least model of T, while program specialization can only replace conjunctions of constraints.

These more powerful rules extend the specialization-based verification techniques in two ways. First, we can verify programs with respect to complex initial and error properties defined by sets of CLP clauses (for instance, recursively defined relations among program variables), whereas program specialization can only deal with initial and error properties specified by (sets of) constraints. Second, we can verify programs which manipulate arrays and other data structures by applying equivalences between predicates that express suitable properties of those structures. In particular, in this paper we will apply equivalences which follow from the axiomatization of the theory of arrays [4].

The paper is organized as follows. In Section 2 we present the imperative language and the definition of its interpreter as a CLP program. In Section 3 we describe how partial correctness properties of imperative programs can be translated to predicates defined by CLP programs. In Section 4 we present our transformation-based verification method, and a general strategy to apply the transformation rules. Next, we present two examples of application of our verification method. In particular, in Section 5 we show how we deal with specifications provided by recursive CLP clauses, and in Section 6 we show how we deal with programs that manipulate arrays. Finally, in Section 7 we discuss related work in the area of program verification.

2 Translating Imperative Programs into CLP

We consider a simple imperative language with arrays. We are given: (i) the set *Vars* of integer variable identifiers, (ii) the set *AVars* of integer array identifiers, and (iii) the set $\mathbb{Z}$ of the integer constants. Our language has the following syntax:

$x, y, z, i, j, m, n, \ldots \in$ *Vars* (integer variable identifiers)
$a, b, \ldots \in$ *AVars* (integer array identifiers)
$k, \ldots \in \mathbb{Z}$ (integer constants)
$\ell, \ell_0, \ell_1, \ldots \in$ Labels ($\subseteq \mathbb{Z}$)
uop, *bop* $\in$ Unary and binary operators (-, +, <, ...)

prog ::= lab_cmd$^+$
lab_cmd ::= ℓ:cmd
cmd ::= `halt` | x=expr | a[expr]=expr | `if (`expr`)` ℓ_1 `else` ℓ_2 | `goto` ℓ
expr ::= k | x | *uop* expr | expr *bop* expr | a[expr]

A program is a non-empty sequence of labeled commands (also called commands, for brevity). The elements of a sequence are separated by semicolons. We assume that in every program each label occurs at most once. Note that in our language we can deal with conditional and iterative commands, such as

'`if (expr) cmd`', '`if (expr) cmd else cmd`', and '`while (expr) {cmd}`', by translating them using `if-else` and `goto` commands.

In order to focus our attention on the verification issues, we do not consider in our imperative language other features such as: (i) type and variable declarations, (ii) function declarations and function calls, and (iii) multidimensional arrays. Those features can be added in a straightforward way (see, for instance, [12]).

Now we give the semantics of our language by defining a binary relation $\Longrightarrow$ which will be encoded by a CLP predicate `tr`. For that purpose let us first introduce the following auxiliary notions.

An *environment* δ is a function that maps: (i) every integer variable identifier x to its value $v \in \mathbb{Z}$, and (ii) every array identifier a to a finite function from the set $\{0,\ldots,dim(a)-1\}$ to $\mathbb{Z}$, where $dim(a)$ is the dimension of a. For any expression e, array identifier a, and environment δ, (i) $[\![e]\!]\delta$ is the integer value of e defined by induction on the structure of e (in particular, for any integer variable identifier x, $[\![x]\!]\delta =_{def} \delta(x)$), and (ii) $[\![a[e]]\!]\delta =_{def} \delta(a)([\![e]\!]\delta)$. We assume that the evaluation of expressions has no side effects.

A *configuration* is a pair of the form $\langle\!\langle c,\delta \rangle\!\rangle$ where: (i) c is a labeled command, and (ii) δ is an environment. By $update(f,x,v)$ we denote the function f' such that, for every y, if $y=x$ then $f'(y)=v$ else $f'(y)=f(y)$. By $write(f,x,v)$ we denote the function $update(f,x,v)$ in the case where f is a finite function denoting an array, x is an integer in the domain of f, and v is an integer value. For any program *prog*, for any label ℓ, (i) $at(\ell)$ denotes the command in *prog* with label ℓ, and (ii) $nextlab(\ell)$ denotes the label of the command in *prog* which is written *immediately after* the command with label ℓ.

The operational semantics (that is, the interpreter) of our language is given as a transition relation $\Longrightarrow$ between configurations according to the following rules $R1$–$R3$. Notice that no rules are given for the command ℓ`:halt`. Thus, no configuration γ exists such that ℓ`:halt` $\Longrightarrow \gamma$.

($R1$). *Assignment.*

If x is an integer variable identifier:

$\langle\!\langle \ell : x=e,\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(nextlab(\ell)),\ update(\delta,x,[\![e]\!]\delta) \rangle\!\rangle$

If a is an integer array identifier:

$\langle\!\langle \ell : a[ie]=e,\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(nextlab(\ell)),\ update(\delta,a,write(\delta(a),[\![ie]\!]\delta,[\![e]\!]\delta)) \rangle\!\rangle$

($R2$). *Goto.* $\langle\!\langle \ell : goto\ \ell',\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell'),\ \delta \rangle\!\rangle$

($R3$). *If-else.*

If $[\![e]\!]\delta \neq 0$: $\langle\!\langle \ell :$ `if (e)` ℓ_1 `else` $\ell_2,\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell_1),\ \delta \rangle\!\rangle$

If $[\![e]\!]\delta = 0$: $\langle\!\langle \ell :$ `if (e)` ℓ_1 `else` $\ell_2,\ \delta \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell_2),\ \delta \rangle\!\rangle$

Let us now recall some notions and terminology concerning constraint logic programming (CLP). For more details the reader may refer to [28]. If $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials with integer coefficients and integer variables, then $\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \neq \mathtt{p}_2$, $\mathtt{p}_1 < \mathtt{p}_2$, $\mathtt{p}_1 \leq \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, and $\mathtt{p}_1 > \mathtt{p}_2$ are *atomic constraints*. A *constraint* is either `true`, or `false`, or an atomic constraint, or a *conjunction* of constraints. A CLP program is a finite set of clauses of the form `A :- c,B`, where `A` is an atom, `c` is a constraint, and `B` is a (possibly empty) conjunction of atoms. A clause of the form: `A :- c` is called a *constrained fact* or simply a *fact* if `c` is `true`.

The semantics of a CLP program P is defined to be the *least model* of P which extends the standard interpretation on the integers. This model is denoted by $M(P)$.

The CLP interpreter for our imperative language is given by the following predicate `tr` which encodes the transition relation $\Longrightarrow$.

1. tr(cf(cmd(L,asgn(int(X),E)),D), cf(cmd(L1,C),D1)) :-
   eval(E,D,V), update(D,int(X),V,D1), nextlab(L,L1), at(L1,C).
2. 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).
3. tr(cf(cmd(L,ite(E,L1,L2)),D), cf(cmd(L1,C),D)) :- V ≠ 0, eval(E,D,V), at(L1,C).
4. tr(cf(cmd(L,ite(E,L1,L2)),D), cf(cmd(L2,C),D)) :- V = 0, eval(E,D,V), at(L2,C).
5. tr(cf(cmd(L,goto(L1)),D), cf(cmd(L1,C),D)) :- at(L1,C).

In the above clauses the term asgn(int(X),E) encodes a variable assignment of the form $x = e$, while asgn(arrayelem(A,IE),E) encodes an array assignment of the form $a[ie] = e$. Similarly, the terms ite(E,L1,L2) and goto(L) encode the conditional if (e) ℓ_1 else ℓ_2 and the jump goto ℓ, respectively. The term cmd(L,C) encodes the command C with label L. The predicate at(L,C) binds to C the command with label L. The predicate nextlab(L,L1) binds to L1 the label of the command which is written immediately after the command with label L.

The predicate eval(E,D,V) computes the value V of the expression E in the environment D. Below we list a subset of the clauses that define eval(E,D,V) by induction on the structure of E. The others are similar and we shall omit them.

6. eval(int(X),D,V) :- lookup(D,int(X),V).
7. eval(not(E),D,V) :- V≠0, V1=0, eval(E,D,V1).
8. eval(not(E),D,V) :- V=0, V1≠0, eval(E,D,V1).
9. eval(plus(E1,E2),D,V) :- V = V1+V2, eval(E1,D,V1), eval(E2,D,V2).
10. eval(arrayelem(A,IE),D,V) :- eval(IE,D,I), lookup(D,array(A),FA), read(FA,I,V).

The predicate update(D,Id,B,D1) updates the environment D, thereby constructing the new environment D1 by binding the (integer or array) identifier Id to the (integer or array) value B. The predicate lookup(D,Id,B) retrieves from the environment D the (integer or array) value B bound to the identifier Id.

The predicate read gets the value of an element of an array, and the predicate write sets the value of an element of an array. As already mentioned, the environment maps an array to a finite function. We represent this function as a pair (A,N), where N is the dimension of the array and A is a list of integers of length N. The predicate read((A,N),I,X) holds iff the I-th element of A is X, and the predicate write((A,N),I,X,(A1,N)) holds iff the list A1 is equal to A except that the I-th element of A1 is X.

For reasons of space, we do not list the clauses defining update, lookup, read, and write.

3 Translating Partial Correctness into CLP

The problem of verifying the partial correctness of a program *prog* is the problem of checking whether or not, starting from an initial configuration, the execution of *prog* leads to an error configuration. This problem is formalized by defining an *incorrectness triple* of the form $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$, where:

(i) *prog* is a program which acts on the variables $z_1, \ldots, z_r$, each of which is either an integer variable or an integer array,

(ii) φ_{init} is a first-order formula with free variables in $\{z_1, \ldots, z_r\}$ characterizing the initial configurations, and

(iii) φ_{error} is a first-order formula with free variables in $\{z_1, \ldots, z_r\}$ characterizing the error configurations.

We assume that: (i) the formulas φ_{init} and φ_{error} can be encoded using CLP predicates, and (ii) the domain of every environment δ is the set $\{z_1,\ldots,z_r\}$. We also assume, without loss of generality, that the last command of *prog* is ℓ_h:`halt` and no other `halt` command occurs in *prog*.

We say that a program *prog* is *incorrect* with respect to a set of initial configurations satisfying φ_{init} and a set of error configurations satisfying φ_{error}, or simply, *prog* is *incorrect* with respect to φ_{init} and φ_{error}, 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 command in *prog*. Obviously, in φ_{init} or in φ_{error} some of the variables $z_1,\ldots,z_r$ may be absent. 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}.

We now show, by means of an example, how to encode an incorrectness triple as a CLP program. The reader will have no difficulty to see how this encoding can be done in general. Let us consider the incorrectness triple $\{\!\{\varphi_{init}(i,j)\}\!\}$ *increase* $\{\!\{\varphi_{error}(i,j)\}\!\}$ where:

– $\varphi_{init}(i,j)$ is $i=0\wedge j=0$,

– *increase* is the program

ℓ_0: `while` $(i<2n)$ {`if` $(i<n)$ $i=i+1$; `else` $j=i+1$; $\quad i=i+1$};

ℓ_h: `halt`

– $\varphi_{error}(i,j)$ is $i<j$.

First, we replace the given program *increase* by the following sequence of commands:

ℓ_0: `if` $(i<2n)$ ℓ_1 `else` ℓ_h;
ℓ_1: `if` $(i<n)$ ℓ_2 `else` ℓ_4;
ℓ_2: $i=i+1$;
ℓ_3: `goto` ℓ_5;
ℓ_4: $j=i+1$;
ℓ_5: $i=i+1$;
ℓ_6: `goto` ℓ_0;
ℓ_h: `halt`

Then, this sequence of commands is translated into the following CLP facts of the form: `at`(ℓ, cmd) meaning that at label ℓ there is the command *cmd*:

1. `at(0,ite(less(int(i),mult(int(2),int(n))),1,h)).`
2. `at(1,ite(less(int(i),int(n)),2,4)).`
3. `at(2,asgn(int(i),plus(int(i),int(1)))).`
4. `at(3,goto(5)).`
5. `at(4,asgn(int(j),plus(int(i),int(1)))).`
6. `at(5,asgn(int(i),plus(int(i),int(1)))).`
7. `at(6,goto(0)).`
8. `at(h,halt).`

We also have the following clauses that specify that an error configuration can be reached from an initial configuration, in which case the atom `incorrect` holds:

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

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

```
12. initConf(cf(cmd(0,ite(less(int(i),mult(int(2),int(n))),1,h)),
          [[int(i),I],[int(j),J],[int(n),N]])) :- phiInit(I,J)
13. errorConf(cf(cmd(h,halt),
          [[int(i),I],[int(j),J],[int(n),N]])) :- phiError(I,J)
14. phiInit(I,J) :- I=0, J=0.
15. phiError(I,J) :- I<J.
```

In the initial configuration (see clause 12) the command (labeled by `0`) is:

`cmd(0,ite(less(int(i),mult(int(2),int(n))),1,h))`

In clauses 12 and 13 the environment δ has been encoded by the list:

`[[int(i),I],[int(j),J],[int(n),N]]`

which provides the bindings for the integer variables *i*, *j*, and *n*, respectively. The initial environment is any environment which binds *i* to 0 and *j* to 0.

The CLP program consisting of clauses 1–15 above, together with the clauses that define the predicate `tr` (see clauses 1–5 of Section 2), is called the *CLP encoding* of the given incorrectness triple $\{\!\{\varphi_{init}(i,j)\}\!\}$ *prog* $\{\!\{\varphi_{error}(i,j)\}\!\}$.

Theorem 1 (Correctness of CLP Encoding) *Let T be the CLP encoding of the incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ prog $\{\!\{\varphi_{error}\}\!\}$. The program prog is correct with respect to φ_{init} and φ_{error} iff* `incorrect` $\notin M(T)$.

4 The Verification Method

In this section we present a method for verifying whether or not a program *prog* is correct with respect to any given pair of φ_{init} and φ_{error} formulas. By Theorem 1, this verification task is equivalent to checking whether or not `incorrect` $\notin M(T)$, where T is the CLP encoding of the incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$.

Our verification method is based on transformations of CLP programs that preserve the least model semantics [15, 17]. It makes use of the following *transformation rules*, collectively called *unfold/fold rules*: *unfolding*, *goal replacement*, *clause removal*, *definition*, and *folding*. The verification method is an extension of the method for proving safety of imperative programs by specialization of CLP programs presented in [12]. Actually, the transformations presented in this paper are much more powerful than program specialization and, as we will show in the next sections, they enable the verification of correctness properties which are more complex than those considered in [12].

Our verification method consists of the following two steps.

Step (A): *Removal of the Interpreter.* The CLP program T which encodes the incorrectness triple $\{\!\{\varphi_{init}\}\!\}$ *prog* $\{\!\{\varphi_{error}\}\!\}$ is specialized with respect to the clauses encoding φ_{init}, *prog*, and φ_{error}. The result of this first transformation step is a new CLP program T_A such that `incorrect` $\in M(T)$ iff `incorrect` $\in M(T_A)$. Program T_A incorporates the operational semantics of the imperative program *prog*, but the clauses defining the predicate `tr`, that is, the interpreter of the imperative language we use, do not occur in T_A, hence the name of this transformation step. Step (A) is common to other verification techniques which are based on program specialization [11, 43].

Step (B): *Propagation of the initial and error properties.* By applying the unfold/fold transformation rules, program T_A is transformed into a new CLP program T_B such that `incorrect` $\in M(T_A)$ iff

incorrect $\in M(T_B)$. This transformation exploits the interactions among the predicates encoding the initial property φ_{init}, the operational semantics of *prog*, and the error property φ_{error}, with the objective of deriving a program T_B where the predicate incorrect is defined either by (i) the fact 'incorrect.', or by (ii) the empty set of clauses (that is, no clauses for incorrect occur in T_B). In Case (i) the imperative program *prog* is incorrect with respect to the given φ_{init} and φ_{error} properties, while in Case (ii) *prog* is correct with respect to these properties. There is a third case where neither (i) nor (ii) holds, that is, in program T_B the predicate incorrect is defined by a non-empty set of clauses not containing the fact 'incorrect.'. In this third case we cannot conclude anything about the correctness of *prog* by a simple inspection of T_B and, similarly to what has been proposed in [12], we iterate Step (B) by propagating at each iteration the initial and error properties (that, in general, could have been modified during the previous iterations), in the hope of deriving a program where either Case (i) or Case (ii) holds. Obviously, due to undecidability limitations, we may never get to one of these two cases.

Note that either Case (i) or Case (ii) may hold for the program T_A that we have derived at the end of Step (A) and, if this happens, we need not perform Step (B) at all.

Step (A) and Step (B) are both instances of the *Transform* strategy presented in Figure 1. These instances are obtained by suitable choices of the *unfolding*, *generalization*, and *goal replacement* auxiliary strategies. Step (A) has been fully automated using the MAP system [38] and always returns a program with no residual call to the predicate tr (this is a consequence of the fact that tr has no recursive calls, and hence all calls to tr can be fully unfolded). A detailed description of Step (A) for a simple C-like imperative language without arrays can be found in [12]. As discussed below, we can also design the unfolding, generalization, and goal replacement auxiliary strategies in such a way that Step (B) always terminates (see, in particular, Theorem 2). However, the design of auxiliary strategies that are effective in practice is a very hard task. Some of those strategies have been automated and, at the moment, we are performing experiments for their evaluation.

Let us briefly illustrate the *Transform* strategy. We assume that the CLP program *P* taken as input by the strategy contains a single clause defining the predicate incorrect of the form: incorrect :- c, G, where c is a constraint (possibly true) and G is a non-empty conjunction of atoms. (The strategy can easily be extended to the case where incorrect is defined by more than one clause.) In particular, we will consider the program *P* made out of: (i) the clauses defining the predicates incorrect and reach (see clauses 9–11 of Section 3), (ii) the clauses defining the predicate tr (see clauses 1–5 of Section 2) which is the interpreter of our imperative language, (iii) the clauses for the predicates initConf and errorConf, and (iv) the clauses defining the predicates on which tr depends (such as at and eval). In *P* the predicate incorrect is defined by the single clause incorrect :- initConf(X), reach(X).

The set of predicate symbols is partitioned into two subsets, called the *high* and *low* predicates, respectively, such that no low predicate depends on a high predicate. Recall that a predicate p *immediately depends on* a predicate q if in the program there is a clause of the form $p(\ldots) :- \ldots, q(\ldots), \ldots$ The *depends on* relation is the transitive closure of the *immediately depends on* relation.

The predicates incorrect, initConf, reach, and errorConf are high predicates and, in general, the body of the clause incorrect :- c, G has at least one occurrence of a high predicate. Moreover, every new predicate introduced during the DEFINITION & FOLDING phase of the *Transform* strategy is a high predicate. This partition is needed for guaranteeing the correctness of the *Transform* strategy (see Theorem 2 below).

The *Transform* strategy makes use of two functions, *Unf* and *Gen*, which are used for performing unfolding and generalization steps, respectively.

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

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

while in *InDefs* there is a clause C *do*
  UNFOLDING:
  $\mathit{TransfC} := \mathit{Unf}(C, A)$, where A is the leftmost atom with high predicate in the body of C;
  while in *TransfC* there is a clause D whose body contains an occurrence of an unfoldable atom B *do*
    $\mathit{TransfC} := (\mathit{TransfC} - \{D\}) \cup \mathit{Unf}(D, B)$
  end-while;
  GOAL REPLACEMENT:
  select a subset R of *TransfC*;
  for every $D \in R$ *do*
    if (i) the constrained goal $\texttt{c}_1, \texttt{G}_1$ occurs in the body of D,
      (ii) all predicates in $\texttt{G}_1$ are low, and
      (iii) $M(P) \models \forall(\texttt{c}_1, \texttt{G}_1 \leftrightarrow \texttt{c}_2, \texttt{G}_2)$,
    then replace $\texttt{c}_1, \texttt{G}_1$ by $\texttt{c}_2, \texttt{G}_2$ in the body of D;
  CLAUSE REMOVAL:
  while in *TransfC* there is a clause F whose body contains an unsatisfiable constraint *do*
    $\mathit{TransfC} := \mathit{TransfC} - \{F\}$
  end-while;
  DEFINITION & FOLDING:
  while in *TransfC* there is a clause E of the form: `H :- e, L, Q, R` such that at least one high predicate occurs in `Q` *do*
    if in *Defs* there is a clause D of the form: `Newd :- d, D`, where:
      (i) for some substitution ϑ, $\texttt{Q} = \texttt{D}\vartheta$, and
      (ii) $\texttt{e} \sqsubseteq \texttt{d}\vartheta$
    then $\mathit{TransfC} := (\mathit{TransfC} - \{E\}) \cup \{\texttt{H :- e, L, Newd}\vartheta\texttt{, R}\}$;
    else let $\mathit{Gen}(E, \mathit{Defs})$ be `Newg :- g, G`, where:
      (i) `Newg` is an atom with high predicate symbol not occurring in $P \cup \mathit{Defs}$,
      (ii) for some substitution ϑ, $\texttt{Q} = \texttt{G}\vartheta$, and
      (iii) $\texttt{e} \sqsubseteq \texttt{g}\vartheta$;
      $\mathit{Defs} := \mathit{Defs} \cup \{\mathit{Gen}(E, \mathit{Defs})\}$; $\mathit{InDefs} := \mathit{InDefs} \cup \{\mathit{Gen}(E, \mathit{Defs})\}$;
      $\mathit{TransfC} := (\mathit{TransfC} - \{E\}) \cup \{\texttt{H :- e, L, Newg}\vartheta\texttt{, R}\}$
  end-while;
  $\mathit{InDefs} := \mathit{InDefs} - \{C\}$; $\mathit{TransfP} := \mathit{TransfP} \cup \mathit{TransfC}$;
end-while;
REMOVAL OF USELESS CLAUSES:
Remove from *TransfP* all clauses whose head predicate is useless.

Figure 1: The *Transform* strategy.

Let us first consider the function *Unf*.

Given a clause C of the form $\mathtt{H}\,\mathtt{:-}\,\mathtt{c},\mathtt{L},\mathtt{A},\mathtt{R}$, 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 $\{\mathtt{K}_i\,\mathtt{:-}\,\mathtt{c}_i,\mathtt{B}_i \mid i = 1,\ldots,m\}$ be the set of the (renamed apart) clauses of program P such that, for $i=1,\ldots,m$, $\mathtt{A}$ is unifiable with $\mathtt{K}_i$ via the most general unifier ϑ_i and $(\mathtt{c},\mathtt{c}_i)\,\vartheta_i$ is satisfiable (thus, the unfolding function performs also some constraint solving operations). We define the following function:

$$\mathit{Unf}(C,\mathtt{A}) = \{(\mathtt{H}\,\mathtt{:-}\,\mathtt{c},\mathtt{c}_i,\mathtt{L},\mathtt{B}_i,\mathtt{R})\,\vartheta_i \mid i = 1,\ldots,m\}$$

Each clause in $\mathit{Unf}(C,\mathtt{A})$ is said to be derived by *unfolding C w.r.t.* $\mathtt{A}$. In order to perform unfolding steps during the execution of the *Transform* strategy, we assume that it is given a *selection function* that determines which atoms occurring in the body of clause C are *unfoldable*. This selection function, which depends on the sequence of applications of *Unf* through which we have derived C, will ensure that any sequence of clauses constructed by unfolding w.r.t. unfoldable atoms is finite. A survey of techniques which ensure the finiteness of unfolding can be found in [37].

The function *Gen*, called the *generalization operator*, is used for introducing new predicate definitions. Indeed, given a clause E and the set *Defs* of clauses defining the predicates introduced so far, $\mathit{Gen}(E,\mathit{Defs})$ returns a new predicate definition G such that E can be *folded* by using clause G. The generalization operator guarantees that during the execution of the *Transform* strategy, a *finite* number of new predicates is introduced (otherwise, the strategy does not terminate). One can define several generalization operators based on the notions of *widening*, *convex hull*, *most specific generalization*, and *well-quasi orderings* which have been introduced for analyzing and transforming constraint logic programs (see, for instance, [7, 9, 13, 19, 42]). For lack of space we do not present here the definitions of those generalization operators, and we refer the reader to the relevant literature.

The equivalences which are considered when performing the goal replacements are called *laws* and their validity in the least model $M(P)$ can be proved once and for all before applying the *Transform* strategy. During the application of the *Transform* strategy we also need an auxiliary strategy for performing the goal replacement steps (this auxiliary strategy has to select the clauses where the replacements should take place and the law to be used). In Section 6 we will consider programs acting on arrays and we will see an application of the goal replacement rule which makes use of a law that holds for arrays. However, we leave it for future work the study of general strategies for performing goal replacement.

In the *Transform* strategy we also use the following notions. We say that a constraint $\mathtt{c}$ *entails* a constraint $\mathtt{d}$, denoted $c \sqsubseteq d$, if $\forall(\mathtt{c}\rightarrow\mathtt{d})$ holds, where, as usual, by $\forall(\varphi)$ we denote the universal closure of a formula φ. The set of *useless predicates* in a program P is the maximal set U of predicates occurring in P such that $\mathtt{p}$ is in U iff every clause with head predicated $\mathtt{p}$ is of the form $\mathtt{p}(\ldots)\ \mathtt{:-}\ \mathtt{c},\mathtt{G}_1,\mathtt{q}(\ldots),\mathtt{G}_2$, for some $\mathtt{q}$ in U. A clause in a program P is *useless* if the predicate of its head is useless in P.

We have the following result.

Theorem 2 (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* $\mathtt{incorrect}\in M(\mathit{TransfP})$.

The termination of the *Transform* strategy is guaranteed by the assumption that one can make only a finite number of unfolding and generalization steps. The correctness of the strategy with respect to the least model semantics directly follows from the correctness of the transformation rules [15, 51]. Indeed, the conditions on high and low predicates ensure that the *Transform* strategy complies with the conditions on predicate *levels* given in [51].

Let us briefly explain how the *Transform* strategy may achieve the objective of deriving a program

TransfP with either the fact '`incorrect.`' (hence proving that *prog* is incorrect) or the empty set of clauses for the predicate `incorrect` (hence proving that *prog* is correct).

If *prog* is incorrect, the fact '`incorrect.`' can, in principle, be derived by unfolding the clause `incorrect :- c, G.` Indeed, observe that if `incorrect` $\in M(P)$ then, by the completeness of the *top-down* evaluation strategy of CLP programs [28], there exists a sequence of unfolding steps that leads to the fact '`incorrect.`'. In practice, however, the ability to derive such a fact depends on the specific strategy used for selecting the atoms for performing unfolding steps during the UNFOLDING phase.

A program *TransfP* where the predicate `incorrect` is defined by the empty set of clauses can be obtained by first deriving a program where the set of clauses defining `incorrect` and the high predicates upon which `incorrect` depends, contains no constrained facts. Indeed, in this case the predicate `incorrect` is useless and all the clauses of its definition are removed by the last step of the *Transform* strategy. A set of clauses without constrained facts may be derived as follows. We perform the UNFOLDING, GOAL REPLACEMENT, and CLAUSE REMOVAL phases as indicated in the *Transform* strategy. If we get a set *TransfC* of clauses with constrained facts, then the strategy will necessarily derive a final program *TransfP* with constrained facts. Otherwise, all clauses in *TransfC* are folded by (possibly) introducing new predicate definitions and the strategy continues by processing the newly introduced predicates (if any). If the strategy exits the while-loop without producing any constrained fact, we derive a program *TransfP* defining a set of mutually recursive predicates without any constrained fact.

Let us consider the incorrectness triple $\{\!\{\varphi_{init}(i,j)\}\!\}$ *increase* $\{\!\{\varphi_{error}(i,j)\}\!\}$ of Section 3. In order to show that the program *increase* is correct with respect to $\varphi_{init} =_{def} i=0 \wedge j=0$ and $\varphi_{error} =_{def} i<j$, we start off from clauses 1–15 associated with the program *increase* (see Section 3), together with the clauses for the predicate `tr` (clauses 1–5 of Section 2 and the clauses on which `tr` depends) which, as already mentioned, define the interpreter of our imperative language.

We perform Step (A) of the verification method by applying the *Transform* strategy.

UNFOLDING. First we unfold clause 9 with respect to the leftmost atom with high predicate, that is, `initConf(X)`, and we get:

16. `incorrect :- I=0, J=0, reach(cf(cmd(0,ite(less(int(i),mult(int(2),int(n))),1,h)), [[int(i),I],[int(j),J],[int(n),N]])).`

DEFINITION & FOLDING. We introduce the new predicate definition:

17. `new1(I,J,N) :- reach(cf(cmd(0,ite(less(int(i),mult(int(2),int(n))),1,h)), [[int(i),I],[int(j),J],[int(n),N]])).`

We fold clause 16 using clause 17 and we get:

18. `incorrect :- I=0, J=0, new1(I,J,N).`

Then, we continue the execution of the *Transform* strategy, starting from the last definition we have introduced, that is, clause 17 (indeed, we have *InDefs* $=\{$clause 17$\}$). Eventually we get the following program T_A:

18. `incorrect :- I=0, J=0, new1(I,J,N).`
19. `new1(I,J,N) :- I<2*N, I<N, I1=I+2, new1(I1,J,N).`
20. `new1(I,J,N) :- I<2*N, I≥N, I1=I+1, J1=I+1, new1(I1,J1,N).`
21. `new1(I,J,N) :- I≥2*N, I<J.`

Now we have completed Step (A).

Unfortunately, it is not possible to prove by direct evaluation that `incorrect` is not a consequence of the above CLP clauses. Indeed, the evaluation of the query `incorrect` using the standard top-down

strategy enters into an infinite loop. Tabled evaluation [10] does not terminate either, as infinitely many tabled atoms are generated. Analogously, bottom-up evaluation is unable to return an answer. Indeed, the presence of a constrained fact for `new1` in program T_A (see clause 21) generates in the least model $M(T_A)$, by repeatedly using the recursive clauses 19 and 20, infinitely many new atoms with predicate `new1`, and thus we cannot show that `new1` does not hold for $\mathtt{I}=0 \wedge \mathtt{J}=0$.

Hence, in this way we cannot show that `incorrect` does not hold in $M(T_A)$ and we cannot conclude that the program *increase* is correct.

Our verification method, instead of directly evaluating the query `incorrect` in T_A, proceeds to Step (B). In order to perform this transformation step we may choose to propagate either the initial property or the error property. Let us opt for the second choice. (However, verification would succeed also by propagating the initial property.) In order to do so, we have first to 'reverse' program T_A, as indicated in [12]. We proceed as follows. First, program T_A is transformed into a program of the form:

s1. `incorrect :- a(U), r1(U).`
s2. `r1(U) :- trans(U,V), r1(V).`
s3. `r1(U) :- b(U).`

where the predicates `a(U)`, `trans(U,V)`, and `b(U)` are defined by the following clauses:

s4. $\mathtt{a}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}]) \texttt{ :- } \mathtt{I}=0,\ \mathtt{J}=0.$
s5. $\mathtt{trans}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}],[\mathtt{new1},\mathtt{I1},\mathtt{J},\mathtt{N}]) \texttt{ :- } \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I1}=\mathtt{I}+2.$
s6. $\mathtt{trans}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}],[\mathtt{new1},\mathtt{I1},\mathtt{J1},\mathtt{N}])\texttt{:- } \mathtt{I}<2*\mathtt{N},\ \mathtt{I}\geq\mathtt{N},\ \mathtt{I1}=\mathtt{I}+1,\ \mathtt{J1}=\mathtt{I}+1.$
s7. $\mathtt{b}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}]) \texttt{ :- } \mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J}.$

This transformation is correct because program T_A can be obtained from clauses s1–s7 by: (i) unfolding clauses s1–s3 with respect to `a(U)`, `trans(U,V)`, and `b(U)`, and then (ii) rewriting the atoms of the form $\mathtt{r1}([\mathtt{new1},\mathtt{X},\mathtt{Y},\mathtt{N}])$ as `new1(X,Y,N)`. (The occurrences of the predicate symbol `new1` in the arguments of `r1` should be considered as an individual constant.) Then, the reversed program T_A^{rev} is given by the following clauses (with the same definitions of `a(U)`, `trans(U,V)`, and `b(U)`):

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

One can show that program reversal is correct in the sense that $\mathtt{incorrect} \in M(T_A)$ iff $\mathtt{incorrect} \in M(T_A^{rev})$ [12].

Now, we perform Step (B) of the verification method. Let us apply the *Transform* strategy taking as input program T_A^{rev} (clauses rev1–rev3) and clauses s4–s7. We assume that the high predicates are: `incorrect`, `b`, `r2`, `trans`, and `a`.

UNFOLDING. First we unfold clause rev1 w.r.t. the leftmost atom with high predicate, that is, `b(U)`, and we get:

22. $\mathtt{incorrect} \texttt{ :- } \mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}]).$

The GOAL REPLACEMENT and the CLAUSE REMOVAL phases leave unchanged the set of clauses we have derived so far.

DEFINITION & FOLDING. In order to fold clause 22 we introduce the clause:

23. $\mathtt{new2}(\mathtt{I},\mathtt{J},\mathtt{N}) \texttt{ :- } \mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}]).$

and we fold clause 22 using clause 23. Thus, we get:

24. $\mathtt{incorrect} \texttt{ :- } \mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{new2}(\mathtt{I},\mathtt{J},\mathtt{N}).$

At this point we execute again the outermost body of the while-loop of the *Transform* strategy because *InDefs* contains clause 23, which is not a constrained fact.

UNFOLDING. By unfolding clause 23 w.r.t. the atom $\mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$, we get the following two clauses:

25. $\mathtt{new2}(\mathtt{I},\mathtt{J},\mathtt{N})$:- $\mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{trans}(\mathtt{U},[\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}]),\ \mathtt{r2}(\mathtt{U})$.
26. $\mathtt{new2}(\mathtt{I},\mathtt{J},\mathtt{N})$:- $\mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{a}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$.

Then, by unfolding clause 25 w.r.t. the atom $\mathtt{trans}(\mathtt{U},[\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$, we get:

27. $\mathtt{new2}(\mathtt{I1},\mathtt{J},\mathtt{N})$:- $\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2\geq 2*\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$.

By unfolding clause 26 w.r.t. $\mathtt{a}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$, we get an empty set of clauses (indeed, the constraint '$\mathtt{I}<\mathtt{J},\mathtt{I}=0,\mathtt{J}=0$' is unsatisfiable).

DEFINITION & FOLDING. In order to fold clause 27 we perform a generalization operation as follows. Clause 27 can be folded introducing the clause:

28. $\mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$:- $\mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2\geq 2*\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$.

However, the comparison between clause 23 introduced in a previous step and clause 28 shows the risk of introducing an unlimited number of definitions whose body contains the atom $\mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$ (see, in particular, the constraint '$\mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J}$' in clause 23 and the constraint '$\mathtt{I}+2\geq 2*\mathtt{N},\ \mathtt{I}+2<\mathtt{J}$' in clause 28), thereby making the *Transform* strategy diverge. To avoid this divergent behaviour, we apply *widening* [9] to clauses 23 and 28, and we introduce the following clause 29, instead of clause 28:

29. $\mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$:- $\mathtt{I}<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$.

Recall that the widening operator applied to two clauses $c1$ and $c2$ (in this order) behaves, in general, as follows. After replacing every equality constraint $\mathtt{A}=\mathtt{B}$ by the equivalent conjunction '$\mathtt{A}\leq\mathtt{B},\ \mathtt{A}\geq\mathtt{B}$', the widening operator returns a clause which is like $c1$, except that the atomic constraints are only those of $c1$ which are implied by the constraint of $c2$. In our case, the widening operator drops the atomic constraint $\mathtt{I}\geq 2*\mathtt{N}$ and keeps $\mathtt{I}<\mathtt{J}$ only.

By folding clause 27 w.r.t. the atom $\mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$, we get:

30. $\mathtt{new2}(\mathtt{I1},\mathtt{J},\mathtt{N})$:- $\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2\geq 2*\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$.

Now, we process the newly introduced definition clause 29 and we perform a new iteration of the body of the outermost while-loop of the *Transform* strategy.

UNFOLDING. After a few unfolding steps from clause 29, we get:

31. $\mathtt{new3}(\mathtt{I1},\mathtt{J},\mathtt{N})$:- $\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{r2}([\mathtt{new1},\mathtt{I},\mathtt{J},\mathtt{N}])$.

DEFINITION & FOLDING. In order to fold clause 31 we do not need to introduce any new definition. Indeed, it is possible to fold clause 31 by using clause 29 and we get:

32. $\mathtt{new3}(\mathtt{I1},\mathtt{J},\mathtt{N})$:-$\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$.

Now, *InDefs* is empty and we exit the outermost while-loop. We get the following program T_B:

24. $\mathtt{incorrect}$:- $\mathtt{I}\geq 2*\mathtt{N},\ \mathtt{I}<\mathtt{J},\ \mathtt{new2}(\mathtt{I},\mathtt{J},\mathtt{N})$.
30. $\mathtt{new2}(\mathtt{I1},\mathtt{J},\mathtt{N})$:- $\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2\geq 2*\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$.
32. $\mathtt{new3}(\mathtt{I1},\mathtt{J},\mathtt{N})$:- $\mathtt{I1}=\mathtt{I}+2,\ \mathtt{I}<2*\mathtt{N},\ \mathtt{I}<\mathtt{N},\ \mathtt{I}+2<\mathtt{J},\ \mathtt{new3}(\mathtt{I},\mathtt{J},\mathtt{N})$.

Since program T_B contains no constrained facts, all its clauses are useless and can be removed from T_B. Thus, at the end of Step (B) we get the final empty program T_B. Hence, $M(T_B)$ is the empty set and the atom $\mathtt{incorrect}$ does not belong to it. We conclude that the given program *increase* is correct with respect to the given properties φ_{init} and φ_{error}.

The various transformation steps which lead to program T_B, including the removal of the interpreter, the program reversal, and the generalization steps performed during Step (B), have been automatically performed by the MAP system [38] (see [12] for some experimental results).

5 Verifying Complex Correctness Properties by Conjunctive Folding

In this section we show how our verification method can be used also in the case when the error properties are specified by sets of CLP clauses, rather than by constraints only. In order to deal with this class of error properties we make use of transformation rules which are more powerful than the ones used in the verification example of the previous section. Indeed, during the DEFINITION&FOLDING phase of the *Transform* strategy, we allow ourselves to introduce new predicates by using definition 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*. Clauses of that form will then be used for performing folding steps. (Note that the new predicate definitions introduced during the verification example of the previous section are all of the form: `Newp :- c, A`, where `A` is a single atom.) The more powerful definition and folding rules, called *conjunctive definition* and *conjunctive folding*, respectively, allow us to perform verification tasks that cannot be handled by the technique presented in [12], which is based on *atomic* definition and *atomic* folding.

Let us consider the following program *gcd* for computing the *greatest common divisor z* of two positive integers *m* and *n*:

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`

and the incorrectness triple $\{\!\{\varphi_{init}(m,n)\}\!\}$ *gcd* $\{\!\{\varphi_{error}(m,n,z)\}\!\}$, where:
– $\varphi_{init}(m,n)$ is $m \geq 1 \wedge n \geq 1$, and
– $\varphi_{error}(m,n,z)$ is $\exists d\,(gcd(m,n,d) \wedge d \neq z)$. The property φ_{error} uses the ternary predicate *gcd* defined by the following CLP clauses:

1. `gcd(X,Y,D) :- X>Y, X1=X-Y, gcd(X1,Y,D).`
2. `gcd(X,Y,D) :- X<Y, Y1=Y-X, gcd(X,Y1,D).`
3. `gcd(X,Y,D) :- X=Y, Y=D.`

Thus, the incorrectness triple holds if and only if, for some positive integers *m* and *n*, the program *gcd* computes a value of *z* that is different from the greatest common divisor of *m* and *n*.

As indicated in Section 4, the program *gcd* can be translated into a set of CLP facts defining the predicate `at`. We will not show them here.

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

4. `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).`
5. `errorConf(cf(cmd(h,halt),`
   `[[int(m),M],[int(n),N],[int(x),X],[int(y),Y],[int(z),Z]])) :- phiError(M,N,Z).`

6. phiInit(M,N) :- M≥1, N≥1.
7. phiError(M,N,Z) :- gcd(M,N,D), D≠Z.

The CLP program encoding the given incorrectness triple consists of clauses 1–7 above, together with the clauses defining the predicate at that encode the program *gcd*, and the clauses that define the predicates incorrect, reach, and tr (that is, clauses 9–11 of Section 3 and clauses 1–5 of Section 2).

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, new1(M,N,M,N,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, gcd(M,N,D), Z≠D.

Clauses 8 and 11 can be rewritten, respectively, as:

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

This rewriting is correct because new1 modifies the values of neither M nor N.

Note that we could avoid performing the above rewriting and automatically derive 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 (see also [43] for a discussion about different styles of encoding the reachability relation and the semantics of imperative languages in CLP).

Now we perform Step (B) of the verification method by applying the *Transform* strategy to the program consisting of clauses $\{1,2,3,8r,9,10,11r\}$. We assume that the only high predicates are incorrect and new1.

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

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

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

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

(Note that by unfolding clause 14 we get an empty set of clauses because the constraints derived in this step are all unsatisfiable.)

The GOAL REPLACEMENT and CLAUSE REMOVAL phases do not modify the set of clauses derived after the UNFOLDING phase because no laws are available for the predicate gcd.

DEFINITION & 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, new1(M,N,X,Y,Z), gcd(X,Y,D).

The body of this clause is the most specific generalization of the bodies of clauses 8r, 15 and 16. Here, we define a conjunction G to be a generalization of a conjunction C if there exists a substitution ϑ such that $G\vartheta$ can be obtained by deleting some of the conjuncts of C.

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

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

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 that contains no constrained facts. Hence both predicates incorrect and new2 are useless and all clauses of *TransfP* can be deleted. Thus, the *Transform* strategy terminates with $TransfP = \emptyset$ and we conclude that the imperative program *gcd* is correct with respect to the given properties φ_{init} and φ_{error}.

6 Verifying Correctness of Array Programs

In this section we apply our verification method of Section 4 for proving properties of a program, called *arraymax*, which computes the maximal element of an array. Let us consider the following incorrectness triple $\{\!\{\varphi_{init}(i,n,a,max)\}\!\}$ *arraymax* $\{\!\{\varphi_{error}(n,a,max)\}\!\}$, where:
– $\varphi_{init}(i,n,a,max)$ is $i=0 \wedge n=dim(a) \wedge n\geq 1 \wedge max=a[i]$,
– *arraymax* is the program

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

– $\varphi_{error}(n,a,max)$ is $\exists k\ (0\leq k<n \wedge a[k]>max)$.

If this triple holds, then the value of *max* computed by the program *arraymax* is not the maximal element of the given array *a* with *n* (≥ 1) elements.

We start off by constructing a CLP program T which encodes the incorrectness triple. This construction is done as indicated in Section 3 and, in particular, the clauses for the predicates phiInit and phiError are generated from the formulas φ_{init} and φ_{error}.

As indicated in Section 4, the program *arraymax* is translated into a set of CLP facts defining the predicate at. The predicates initConf and errorConf 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).

Next, we apply Step (A) of our verification method which consists in removing the interpreter from program T. By applying the *Transform* strategy as indicated in Section 4, we obtain the following program T_A:

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).

We have that new1(I,N,A,Max) encodes the reachability of the error configuration from a configuration where the program variables i,n,a,max are bound to I,N,A,Max, respectively.

In order to propagate the error property, similarly to the example of Section 4, we first 'reverse' program T_A and we get the following program T_A^{rev}:

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

where predicates a(U), b(U), and trans(U,V) 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).

For the application of Step (B) of the verification method we assume that the predicates incorrect, b, r2, trans, and a are high predicates, while the predicate read is a low predicate.

For the GOAL REPLACEMENT phase we use the following law, which is a consequence of the fact that an array is a finite function:

(Law L1) read((A,N),K,Z), read((A,N),I,M) ↔
(K=I, Z=M, read((A,N),K,Z)) ∨
(K≠I, read((A,N),K,Z), read((A,N),I,M))

In general, when applying the *Transform* strategy in the case of array programs, some additional laws may be required (see, for instance, [4]). For the DEFINITION & FOLDING phase we use a particular generalization operator, called *WidenSum* [19], which is a variant of the classical widening operator [7] and behaves as follows.

Given any atomic constraint a, let us denote by $sumcoeff(a)$ the sum of the absolute values of the coefficients of a. Given any two constraints c and d, $WidenSum(c,d)$ returns the conjunction of: (i) all atomic constraints a in c such that $d \sqsubseteq a$, and (ii) all atomic constraints b in d such that $sumcoeff(b) \leq sumcoeff(e)$ for some atomic constraint e in c.

Let us now apply Step (B) of the verification method.

UNFOLDING. We start off by unfolding 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]).

The GOAL REPLACEMENT and CLAUSE REMOVAL phases leave unchanged the set of clauses we have derived so far.

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 clause 10 is in *InDefs*.

UNFOLDING. By unfolding clause 10 w.r.t. the atom r2([new1,I,N,A,Max]), we get the following clauses:

12. new2(I,N,A,Max,K,Z) :- I≥N, K≥0, K<N, Z>Max,
read((A,N),K,Z), trans(U,[new1,I,N,A,Max]), r2(U).
13. new2(I,N,A,Max,K,Z) :- I≥N, K≥0, K<N, Z>Max, read((A,N),K,Z), a([new1,I,N,A,Max]).

By unfolding clause 12 w.r.t. trans(U,[new1,I,N,A,Max]), we get:

14. 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]).
15. 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]).

By unfolding clause 13 w.r.t. a([new1,I,N,A,Max]), we get an empty set of clauses (indeed, the constraint I≥N, I=0, N≥1 is unsatisfiable).

GOAL REPLACEMENT. This phase performs the following two steps: (i) it replaces the conjunction of atoms 'read((A,N),K,Z), read((A,N),I,M)' occurring in the body of clause 14 by the right hand side of Law L1, and then (ii) it splits the derived clause into the following two clauses, each of which corresponds to a disjunct of that right hand side.

14.1 new2(I1,N,A,M,K,Z) :- I1=I+1, N=I1, K≥0, K<I1, M>Max, Z>M,
K=I, Z=M, read((A,N),K,Z), r2([new1,I,N,A,Max]).
14.2 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, Z=M' in the body of clause 14.1 is unsatisfiable. Therefore, this clause is removed from *TranfP*. From clause 14.2, by replacing 'K≠I' by 'K<I ∨ K>I' and simplifying the constraints, we get:

16. 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 performing from clause 15 a sequence of goal replacement and clause removal transformations similar to that we have performed from clause 14, we get the following clause:

17. 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. The comparison between the definition clause 10 we have introduced above, and clauses 16 and 17 which we should fold, shows the risk of introducing an unlimited number of definitions whose body contains the atoms read((A,N),K,Z) and r2([new1,I,N,A,Max]). Thus, in order to fold clauses 16 and 17, we introduce the following new definition:

18. 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]).

The constraint in the body of this clause is obtained by generalizing: (i) the projection of the constraint in the body of clause 16 on the variables I,N,A,Max,K,Z (which are the variables of clause 16 that occur in the atoms read((A,N),K,Z) and r2([new1,I,N,A,Max])), and (ii) the constraint occurring in the body of clause 10. This generalization step can be seen as an application of the above mentioned

WidenSum generalization operator. The same definition clause 18 could also be derived by generalizing the projection of the constraint in the body of clause 16 (instead of 17) and the constraint occurring in the body of clause 10.

Thus, by folding clause 16 and clause 17 using clause 18 we get:

19. $\mathtt{new2(I1,N,A,Max,K,Z)}$:- $\mathtt{I1=I+1,\ N=I1,\ K\geq 0,\ K<I,\ M>Max,\ Z>M,}$
$\mathtt{read((A,N),I,M),\ new3(I,N,A,Max,K,Z)}$.
20. $\mathtt{new2(I1,N,A,M,K,Z)}$:- $\mathtt{I1=I+1,\ N=I1,\ K\geq 0,\ K<I,\ M\leq Max,\ Z>Max,}$
$\mathtt{read((A,N),I,M),\ new3(I,N,A,Max,K,Z)}$.

Now we perform the third iteration of the body of the while-loop of the strategy.

UNFOLDING, GOAL REPLACEMENT, and CLAUSE REMOVAL. By unfolding, goal replacement, and clause removal, from clause 18 we get:

21. $\mathtt{new3(I1,N,A,M,K,Z)}$:- $\mathtt{I1=I+1,\ K\geq 0,\ K<I,\ N\geq I1,\ M>Max,\ Z>M,}$
$\mathtt{read((A,N),K,Z),\ read((A,N),I,M),\ r2([new1,I,N,A,Max])}$.
22. $\mathtt{new3(I1,N,A,Max,K,Z)}$:- $\mathtt{I1=I+1,\ K\geq 0,\ K<I,\ N\geq I1,\ M\leq Max,\ Z>Max,}$
$\mathtt{read((A,N),K,Z),\ read((A,N),I,M),\ r2([new1,I,N,A,Max])}$.

DEFINITION & FOLDING. In order to fold clauses 21 and 22, we do not need to introduce any new definition. Indeed, it is possible to fold these clauses by using clause 18, thereby obtaining:

23. $\mathtt{new3(I1,N,A,M,K,Z)}$:- $\mathtt{I1=I+1,\ K\geq 0,\ K<I,\ N\geq I1,\ M>Max,\ Z>M,}$
$\mathtt{read((A,N),I,M),\ new3(I,N,A,Max,K,Z)}$.
24. $\mathtt{new3(I1,N,A,Max,K,Z)}$:- $\mathtt{I1=I+1,\ K\geq 0,\ K<I,\ N\geq I1,\ M\leq Max,\ Z>Max,}$
$\mathtt{read((A,N),I,M),\ new3(I,N,A,Max,K,Z)}$.

Since no clause to be processed is left (because $\mathit{InDefs}=\emptyset$), the *Transform* strategy exits the outermost while-loop, and the program derived is the set $\{11,19,20,23,24\}$ of clauses. No clause in this set is a constrained fact, and hence by REMOVAL OF USELESS CLAUSES we get the final program T_B consisting of the empty set of clauses. Thus, *arraymax* is correct with respect to the given properties φ_{init} and φ_{error}.

7 Related Work and Conclusions

The verification framework introduced in this paper is an extension of the framework presented in [12], where CLP and iterated specialization have been used to define a general verification framework which is parametric with respect to the programming language and the logic used for specifying the properties of interest. The main novelties we have introduced in this paper are the following: (i) we consider imperative programs acting also on array variables, and (ii) we consider a more expressive specification language, which allows us to write properties involving elements of arrays and, in general, fields of complex data structures.

In order to deal with these additional features (i) we have defined the operational semantics for array manipulation, and (ii) we have considered powerful transformation rules, such as conjunctive definition, conjunctive folding, and goal replacement. The transformation rules and some strategies for their application have been implemented in the MAP transformation system [38], so as to perform proofs of program correctness in a semi-automated way.

Our approach has many connections to various techniques developed in the field of static program analysis, to which David Schmidt has given an outstanding contribution, specially in clarifying its relationships with methodologies like denotational semantics, abstract interpretation, and model checking (see, for example, [48]).

Now we briefly overview the verification techniques that are particularly relevant to the method described in the present paper, and make use of logic programming, constraints, abstract interpretation, and automated theorem proving.

The use of logic programming techniques for program analysis is not novel. For instance, Datalog (the function-free fragment of logic programming) has been proposed for performing various types of program analysis such as *dataflow analysis*, *shape analysis*, and *points-to analysis* [5, 45, 52]. In order to encode the properties of interest into Datalog, all these analysis techniques make an abstraction of the program semantics. In contrast, our transformational method manipulates a CLP program which encodes the full semantics (up to a suitable level of detail) of the program to be analyzed. An advantage of our approach is that the output of a transformation-based analysis is a new program which is *equivalent* to the initial one, and thus the analysis can be iterated to the desired degree of precision.

Constraint logic programming has been successfully applied to perform model checking of both finite [41] and infinite state systems [14, 16, 19]. Indeed, CLP turns out to be suitable for expressing both (i) the symbolic executions and (ii) the invariants of imperative programs [30]. Moreover, there are powerful CLP-based tools, such as ARMC [44], TRACER [29], and HSF [23] 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 [12] 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 with respect to the one of the above mentioned tools.

The connection between imperative programs and constraint logic programming has been investigated in [20, 43]. The verification method presented in [20] is based on a semantics preserving translation from an imperative language with dynamic data structures and recursive functions into CLP. This translation reduces the verification of the (in)correctness of imperative programs to a problem of constraint satisfiability within standard CLP systems. A method based on specialization of CLP programs for performing the analysis of imperative programs has been presented in [43]. In this work the authors first present a CLP interpreter which defines the operational semantics of a simple imperative language. Then, given an imperative program, say *prog*, they specialize that interpreter with respect to a CLP translation of *prog* thereby getting a residual, specialized CLP program P_{sp}. Finally, a static analyzer for CLP programs is applied to P_{sp} and its results are used to annotate the original imperative program with invariants. In [27] a very similar methodology is applied for the verification of low-level programs for PIC microcontrollers.

The program specialization which is done during Step (A) of our verification method (that is, the removal of the interpreter) is very similar to the specialization proposed in [43]. However, having removed the interpreter, in order to verify the correctness of the given imperative program, in Step (B) we apply again program transformation and we not use static analyzers.

CLP approaches to the verification of program correctness have recently received a growing attention because of the development of very efficient constraint solving tools [46]. These approaches include: (i) the template-based invariant synthesis [2], and (ii) the interpolant generation [47]. Related to this line of work, we would like to mention the paper [24] where the authors propose a method for constructing verification tools starting from a given set of proof rules, called Horn-like rules, specified by using constraints.

Our transformational method for verifying properties of array programs is related to several methods based on abstract interpretation and predicate abstraction. In [26], which builds upon [22], relational properties among array elements are discovered by partitioning the arrays into symbolic slices and associating an abstract variable with each slice. This approach offers a compromise between the efficiency

of *array smashing* (where the whole array is represented by a single abstract variable) and the precision of *array expansion* [3] (where every array element is associated with a distinct abstract variable). In [8] the authors present a scalable, parameterized abstract interpretation framework for the automatic analysis of array programs based on slicing. In [25] a powerful technique using template-based quantified abstract domains, is applied to successfully generate quantified invariants. Other authors (see [21, 35]) use indexed predicate abstraction for inferring universally quantified properties about array elements.

Also theorem provers have been used for discovering invariants in programs which manipulate arrays. 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 [32, 34, 39] the authors present some theorem provers which may generate array invariants and interpolants. In [49] a backward reachability analysis based on predicate abstraction and the CEGAR technique is used for deriving invariants which are universally quantified formulas over array indexes, and existing techniques for quantifier-free assertions are adapted to the verification of array properties.

Finally, we would like to mention two more papers which deal with the problem of verifying properties of array programs by using Satisfiability Modulo Theory (SMT) techniques. Paper [36] presents a constraint-based invariant generation method for generating universally quantified loop invariants over arrays, and paper [1] proposes a model checker for verifying universally quantified safety properties of arrays.

As future work, we plan to investigate the issue of designing a general, fully automated strategy for guiding the application of the program transformation rules we have considered in this paper. In particular, we want to study the problem of devising unfolding strategies, generalization operators, and goal-replacement techniques which are tailored to the specific task of verifying program correctness. We also intend to extend the implementation described in [12] and to perform more experiments for validating our transformation-based verification approach by using some benchmark suites which include array programs.

As a further line of future research, we plan to enrich our framework by considering some more theories, besides the theory of the arrays, so that one can prove properties of programs acting on dynamic data structures such as lists and heaps.

8 Acknowledgments

We thank the anonymous referees for their constructive comments. We would like to thank Anindya Banerjee, Olivier Danvy, Kyung-Goo Doh, and John Hatcliff for their kind invitation to contribute to this symposium in honor of Dave Schmidt. Alberto recalls with great joy and gratitude the time together with Dave in Edinburgh and in Rome.

References

[1] F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise & N. Sharygina (2012): *SAFARI: SMT-based Abstraction For Arrays with Interpolants*. In: *Proceedings of the 24th International Conference on Computer Aided Verification, CAV '12*, Lecture Notes in Computer Science 7358, Springer, pp. 679–685, doi:10.1007/978-3-642-31424-7_49.

[2] D. Beyer, T. A. Henzinger, R. Majumdar & A. Rybalchenko (2007): *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, Springer, pp. 378–394, doi:10.1007/978-3-540-69738-1_27.

[3] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux & X. Rival (2002): *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. In: *The Essence of Computation*, Lecture Notes in Computer Science 2566, Springer, pp. 85–108, doi:10.1007/3-540-36377-7_5.

[4] Aaron R. Bradley, Zohar Manna & Henny B. Sipma (2006): *What's decidable about arrays?* In: *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation. VMCAI '06*, Lecture Notes in Computer Science 3855, Springer, pp. 427–442, doi:10.1007/11609773_28.

[5] M. Bravenboer & Y. Smaragdakis (2009): *Strictly declarative specification of sophisticated points-to analyses*. In S. Arora & G. T. Leavens, editors: *Proceedings of the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '09*, ACM, pp. 243–262, doi:10.1145/1640089.1640108.

[6] R. M. Burstall & J. Darlington (1977): *A Transformation System for Developing Recursive Programs*. *Journal of the ACM* 24(1), pp. 44–67, doi:10.1145/321992.321996.

[7] P. Cousot & R. Cousot (1977): *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*, ACM, pp. 238–252, doi:10.1145/512950.512973.

[8] P. Cousot, R. Cousot & F. Logozzo (2011): *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*, ACM, pp. 105–118, doi:10.1145/1926385.1926399.

[9] P. Cousot & N. Halbwachs (1978): *Automatic Discovery of Linear Restraints among Variables of a Program*. In: *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, POPL '78*, ACM, pp. 84–96, doi:10.1145/512760.512770.

[10] B. Cui & D. S. Warren (2000): *A System for Tabled Constraint Logic Programming*. In J. W. Lloyd, editor: *Proceedings of the First International Conference on Computational Logic, CL 2000, London, UK, July 24-28, 2000*, Lecture Notes in Artificial Intelligence 1861, Springer-Verlag, pp. 478–492, doi:10.1007/3-540-44957-4_32.

[11] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2013): *Specialization with Constrained Generalization for Software Model Checking*. In: *Proceedings of the 22nd International Symposium Logic-Based Program Synthesis and Transformation, LOPSTR '12*, Lecture Notes in Computer Science 7844, Springer, pp. 51–70, doi:10.1007/978-3-642-38197-3_5.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi & M. Proietti (2013): *Verifying Programs via Iterated Specialization*. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM '13*, ACM, pp. 43–52, doi:10.1145/2426890.2426899.

[13] D. De Schreye, R. Glück, J. Jørgensen, M. Leuschel, B. Martens & M. H. Sørensen (1999): *Conjunctive Partial Deduction: Foundations, Control, Algorithms, and Experiments*. *Journal of Logic Programming* 41(2–3), pp. 231–277, doi:10.1016/S0743-1066(99)00030-8.

[14] G. Delzanno & A. Podelski (1999): *Model Checking in CLP*. In R. Cleaveland, editor: *5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, Lecture Notes in Computer Science 1579, Springer-Verlag, pp. 223–239, doi:10.1007/3-540-49059-0_16.

[15] S. Etalle & M. Gabbrielli (1996): *Transformations of CLP Modules*. *Theoretical Computer Science* 166, pp. 101–146, doi:10.1016/0304-3975(95)00148-4.

[16] F. Fioravanti, A. Pettorossi & M. Proietti (2001): *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, University of Southampton, UK, pp. 85–96.

[17] F. Fioravanti, A. Pettorossi & M. Proietti (2004): *Transformation Rules for Locally Stratified Constraint Logic Programs*. In K.-K. Lau & M. Bruynooghe, editors: *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, Springer-Verlag, pp. 292–340, doi:10.1007/978-3-540-25951-0_10.

[18] F. Fioravanti, A. Pettorossi, M. Proietti & V. Senni (2011): *Improving Reachability Analysis of Infinite State Systems by Specialization*. In G. Delzanno & I. Potapov, editors: *Proceedings of the 5th International Workshop on Reachability Problems, RP '11, September 28-30, 2011, Genova, Italy*, Lecture Notes in Computer Science 6945, Springer, pp. 165–179, doi:10.1007/978-3-642-24288-5_15.

[19] F. Fioravanti, A. Pettorossi, M. Proietti & V. Senni (2013): *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), pp. 175–199, doi:10.1017/S1471068411000627.

[20] C. Flanagan (2004): *Automatic software model checking via constraint logic*. *Sci. Comput. Program.* 50(1–3), pp. 253–270, doi:10.1016/j.scico.2004.01.006.

[21] C. Flanagan & S. Qadeer (2002): *Predicate abstraction for software verification*. In: *Proceedings of the 29th ACM Symposium on Principles of programming languages, POPL '02*, ACM, pp. 191–202, doi:10.1145/503272.503291.

[22] D. Gopan, T. W. Reps & S. Sagiv (2005): *A framework for numeric analysis of array operations*. In: *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of programming languages, POPL '05*, ACM, pp. 338–350, doi:10.1145/1047659.1040333.

[23] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *HSF(C): A Software Verifier based on Horn Clauses*. In C. Flanagan & B. König, editors: *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, Springer, pp. 549–551, doi:10.1007/978-3-642-28756-5_46.

[24] S. Grebenshchikov, N. P. Lopes, C. Popeea & A. Rybalchenko (2012): *Synthesizing software verifiers from proof rules*. In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, ACM, pp. 405–416, doi:10.1145/2345156.2254112.

[25] B. S. Gulavani, S. Chakraborty, A. V. Nori & S. K. Rajamani (2008): *Automatically Refining Abstract Interpretations*. In: *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '08*, Lecture Notes in Computer Science 4963, Springer, pp. 443–458, doi:10.1007/978-3-540-78800-3_33.

[26] N. Halbwachs & M. Péron (2008): *Discovering properties about arrays in simple programs*. In: *Proceedings of the ACM Conference on Programming language design and implementation, PLDI '08*, ACM, pp. 339–348, doi:10.1145/1375581.1375623.

[27] K. S. Henriksen & J. P. Gallagher (2006): *Abstract Interpretation of PIC Programs through Logic Programming*. In: *Proceedings of the 6th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM '06*, IEEE, pp. 103 – 179, doi:10.1016/0743-1066(92)90030-7.

[28] J. Jaffar & M. Maher (1994): *Constraint Logic Programming: A Survey*. *Journal of Logic Programming* 19/20, pp. 503–581, doi:10.1016/0743-1066(94)90033-7.

[29] J. Jaffar, J. A. Navas & A. E. Santosa (2012): *TRACER: A Symbolic Execution Tool for Verification*, doi:10.1007/978-3-642-31424-7_61. `http://paella.d1.comp.nus.edu.sg/tracer/`.

[30] J. Jaffar, J. A. Navas & A. E. Santosa (2012): *Unbounded Symbolic Execution for Program Verification*. In: *Proceedings of the 2nd International Conference on Runtime Verification, RV '11*, Lecture Notes in Computer Science 7186, Springer, pp. 396–411, doi:10.1007/978-3-642-29860-8_32.

[31] R. Jhala & R. Majumdar (2009): *Software model checking*. *ACM Computing Surveys* 41(4), pp. 21:1–21:54, doi:10.1145/1592434.1592438.

[32] R. Jhala & K. L. McMillan (2007): *Array abstractions from proofs*. In: *Proceedings of the 19th International Conference on Computer Aided Verification, CAV '07*, Lecture Notes in Computer Science 4590, Springer, pp. 193–206, doi:10.1007/978-3-540-73368-3_23.

[33] N. D. Jones, C. K. Gomard & P. Sestoft (1993): *Partial Evaluation and Automatic Program Generation*. Prentice Hall.

[34] L. Kovács & A. Voronkov (2009): *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 '09, Lecture Notes in Computer Science 5503, Springer, pp. 470–485, doi:10.1007/978-3-642-00593-0_33.

[35] S. K. Lahiri & R. E. Bryant (2007): *Predicate abstraction with indexed predicates*. *ACM Trans. Comput. Log.* 9(1), 4, ACM, 29 pages, doi:10.1145/1297658.1297662.

[36] D. Larraz, E. Rodríguez-Carbonell & A. Rubio (2013): *SMT-Based Array Invariant Generation*. In: *14th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '13, Rome, Italy, January 20-22, 2013*, Lecture Notes in Computer Science 7737, Springer, pp. 169–188, doi:10.1007/978-3-642-35873-9_12.

[37] M. Leuschel & M. Bruynooghe (2002): *Logic program specialisation through partial deduction: Control issues*. *Theory and Practice of Logic Programming* 2(4&5), pp. 461–515, doi:10.1017/S147106840200145X.

[38] 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`.

[39] K. L. McMillan (2008): *Quantified invariant generation using an interpolating saturation prover*. In: *Proceedings of 14th international conference on Tools and algorithms for the construction and analysis of systems, TACAS '08*, Lecture Notes in Computer Science 4963, Springer, pp. 413–427, doi:10.1007/978-3-540-78800-3_31.

[40] S. P. Miller, M. W. Whalen & D. D. Cofer (2010): *Software model checking takes off*. *Commun. ACM* 53(2), ACM, pp. 58–64, doi:10.1145/1646353.1646372.

[41] U. Nilsson & J. Lübcke (2000): *Constraint Logic Programming for Local and Symbolic Model-Checking*. In J. W. Lloyd, editor: *Proceedings of the First International Conference on Computational Logic, CL 2000, London, UK, July 24-28, 2000*, Lecture Notes in Artificial Intelligence 1861, Springer-Verlag, pp. 384–398, doi:10.1007/3-540-44957-4_26.

[42] J. C. Peralta & J. P. Gallagher (2003): *Convex Hull Abstractions in Specialization of CLP Programs*. In M. Leuschel, editor: *Logic Based Program Synthesis and Tranformation, 12th International Workshop, LOPSTR '02, Madrid, Spain, September 17–20, 2002, Revised Selected Papers*, Lecture Notes in Computer Science 2664, Springer, pp. 90–108, doi:10.1007/3-540-45013-0_8.

[43] J. C. Peralta, J. P. Gallagher & H. Saglam (1998): *Analysis of Imperative Programs through Analysis of Constraint Logic Programs*. In G. Levi, editor: *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, Springer, pp. 246–261, doi:10.1007/3-540-49727-7_15.

[44] A. Podelski & A. Rybalchenko (2007): *ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement*. In M. Hanus, editor: *Practical Aspects of Declarative Languages, PADL '07*, Lecture Notes in Computer Science 4354, Springer, pp. 245–259, doi:10.1007/978-3-540-69611-7_16.

[45] T. W. Reps (1998): *Program analysis via graph reachability*. *Information and Software Technology* 40(11–12), pp. 701–726, doi:10.1016/S0950-5849(98)00093-7.

[46] A. Rybalchenko (2010): *Constraint Solving for Program Verification: Theory and Practice by Example*. In T. Touili, B. Cook & P. Jackson, editors: *Proceedings of the 22nd International Conference on Computer Aided Verification, CAV '10*, Lecture Notes in Computer Science 6174, Springer, pp. 57–71, doi:10.1007/978-3-642-14295-6_7.

[47] A. Rybalchenko & V. Sofronie-Stokkermans (2010): *Constraint solving for interpolation*. *Journal of Symbolic Computation* 45(11), pp. 1212–1233, doi:10.1007/978-3-540-69738-1_25.

[48] D. A. Schmidt (1998): *Data flow analysis is model checking of abstract interpretations*. In: *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98*, ACM, pp. 38–48, doi:10.1145/268946.268950.

[49] M. N. Seghir, A. Podelski & T. Wies (2009): *Abstraction Refinement for Quantified Array Assertions*. In: *Proceeding of the 16th International Symposium on Static Analysis, SAS '09*, Lecture Notes in Computer Science 5673, Springer, pp. 3–18, doi:10.1007/978-3-642-03237-0_3.

[50] H. Tamaki & T. Sato (1984): *Unfold/Fold Transformation of Logic Programs*. In S.-Å. Tärnlund, editor: *Proceedings of the Second International Conference on Logic Programming, ICLP '84*, Uppsala University, Uppsala, Sweden, pp. 127–138.

[51] H. Tamaki & T. Sato (1986): *A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation*. Technical Report 86-4, Ibaraki University, Japan.

[52] J. Whaley & M. S. Lam (2004): *Cloning-based context-sensitive pointer alias analysis using binary decision diagrams*. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI '04*, ACM, pp. 131–144, doi:10.1145/996841.996859.