Program Verification via Iterated Specialization[☆]

E. De Angelis[a,*], F. Fioravanti[a,*], A. Pettorossi[b,*], M. Proietti[c,*]

[a]*DEC, University "G. d'Annunzio", Viale Pindaro 42, 65127 Pescara, Italy*
[b]*DICII, University of Rome Tor Vergata, Via del Politecnico 1, 00133 Roma, Italy*
[c]*CNR-IASI, Viale Manzoni 30, 00185 Roma, Italy*

Abstract

We present a method for verifying properties of imperative programs by using techniques based on the specialization of constraint logic programs (CLP). We consider a class of imperative programs with integer variables and we focus our attention on safety properties, stating that no error configuration can be reached from any initial configuration. We introduce a CLP program I that encodes the interpreter of the language and defines a predicate `unsafe` equivalent to the negation of the safety property to be verified. Then, we specialize the CLP program I with respect to the given imperative program and the given initial and error configurations, with the objective of deriving a new CLP program I_{sp} that either contains the fact `unsafe` (and in this case the imperative program is proved unsafe) or contains no clauses with head `unsafe` (and in this case the imperative program is proved safe). If I_{sp} enjoys neither of these properties, we iterate the specialization process with the objective of deriving a CLP program where we can prove unsafety or safety. During the various specializations we may apply different strategies for propagating information (either propagating forward from an initial configuration to an error configuration, or propagating backward from an error configuration to an initial configuration) and different operators (such as the widening and the convex hull operators) for generalizing predicate definitions. Each specialization step is guaranteed to terminate, but due to the undecidability of program safety, the iterated specialization process may not terminate. By an experimental evaluation carried out on a significant set of examples taken from the literature, we show that our method improves the precision of program verification with respect to state-of-the-art software model checkers.

Keywords: software model checking, constraint logic programming, program specialization, program transformation

1. Introduction

Formal verification of software products is gaining more and more attention as a promising methodology for increasing the reliability and reducing the cost of software production (see [37] for some case studies). In particular, *software model checking* has the goal of performing formal software verification by combining and extending techniques developed in the fields of static program analysis and model checking (see [32] for a recent survey).

In this paper we consider programs acting on integer variables written in a subset of the imperative C Intermediate Language [38]. Then we address the problem of verifying *safety* properties, stating that when executing a program, an error configuration cannot be reached from any initial configuration.

Since for programs that act on integer variables the safety problem is undecidable, many program analysis techniques follow approaches based on *abstraction* [7], by which the integer data domain is mapped into an abstract domain so that reachability is preserved, in the sense that if a concrete configuration is reachable, then the corresponding abstract configuration is reachable. By a suitable choice of the abstract domain one can design reachability algorithms that terminate and, whenever they prove that an abstract error configuration is not reachable from any

[☆]This paper is an extended, improved version of [13].
[*]Corresponding author
Email addresses: `emanuele.deangelis@unich.it` (E. De Angelis), `fioravanti@unich.it` (F. Fioravanti), `adp@iasi.cnr.it` (A. Pettorossi), `proietti@iasi.cnr.it` (M. Proietti)

abstract initial configuration, then the program is proved to be safe (see [32] for a general abstract reachability algorithm). Notable abstractions are those based on convex polyhedra [10], that is, conjunctions of linear inequalities (also called *constraints* in this paper).

Constraint Logic Programming (CLP) is a very suitable framework for the analysis of imperative programs, because it provides a very convenient way of representing symbolic program executions and also, by using constraints, program abstractions (see, for instance, [29, 31, 40, 41]). In the context of CLP-based program analysis, the problem of verifying imperative programs can be transformed into the problem of analyzing CLP programs by using CLP program specialization [40]. By following the approach presented in [40], first the semantics of the language in which the imperative programs are written is defined in terms of a CLP program I which is the interpreter of that language, and then I is specialized with respect to: (i) the CLP representation P' of the input imperative program P, and (ii) the CLP representation F' of the safety property F that should be checked. The result of this specialization is a CLP program I_{sp} that is semantically equivalent to $I \cup P' \cup F'$. Thus, in order to prove some given properties of the imperative program P, we can analyze the CLP program I_{sp} by applying, for instance, techniques based on the above mentioned polyhedral abstractions.

It has also been pointed out that CLP program specialization can be used as a technique for software model checking on its own [12]. Indeed, by specializing I_{sp} with respect to the constraints characterizing the input values of P (that is, the precondition of P), in some cases one can derive a new CLP program I'_{sp} whose least model $M(I'_{sp})$ can be computed in finite time because I'_{sp} can be represented by a finite (possibly empty) set of constraints. Thus, in these cases it is possible to verify whether or not P is safe by simply inspecting that model.

However, due to the undecidability of safety, it is impossible to devise a specialization technique that always terminates and produces a specialized CLP program whose least model can be finitely computed. Thus, the best one can do is to propose a verification technique based on some heuristics and show that it works well in practice. This is what we have done in this paper. In particular, we have proposed a method, called the *iterated specialization*, that is based on the repeated application of program specialization. By iterated specialization we can produce a sequence of CLP programs of the form $I, I_{sp}, I^1_{sp}, I^2_{sp}, \ldots$ Each program specialization step terminates and has the effect of modifying the structure of the CLP program and explicitly adding new constraints that denote invariants of the computation. Thus, the effect of the iterated specialization is the propagation of these constraints from one program version to the next, and since each new iteration starts from the output of the previous one, we can refine program analysis and possibly increase the level of precision. Iterated specialization terminates at step k, if a *lightweight analysis* based on a simple inspection of the CLP program I^k_{sp} is able to decide safety or unsafety.

In order to validate the heuristics used in our verification method from an experimental point of view, we have implemented a prototype verification system called VeriMAP [15]. We have performed verification tests on a significant set of over 200 programs taken from various publicly available benchmarks. The precision of our system, that is the ratio of successfully verified programs over the total number of programs, is about 85 percent. We have also compared the results we have obtained using the VeriMAP system with the results we have obtained using other state-of-the-art software model checking systems, such as ARMC [41], HSF(C) [23], and TRACER [30]. These results show that our verification system has a considerably higher precision.

Let us summarize here the main contributions of our paper.

(i) The adaptation and the integration of various techniques for specializing and transforming constraint logic programs into the novel iterated specialization method for verifying imperative programs. This adaptation has required:

(i.1) The customization of general purpose unfolding and generalization strategies (such as those in [17, 39]) to the specific task of specializing the interpreter of the programming language under consideration, as well as the CLP programs derived from the interpreter in subsequent iterations of the method. In particular, we have adapted to our context suitable strategies, based on operators often used in static program analysis such as *widening* and *convex hull* [7, 10], for the automatic discovery of loop invariants of the imperative programs to be verified.

(i.2) The customization of general purpose transformations for inverting the order of computation [5], so that iterated program specialization can exploit in a particularly good way the information present in the initial and in the error configurations.

(ii) The implementation of our iterated specialization method into a prototype automatic tool [15].

(iii) The experimental evaluation of our proposed method and its comparison with respect to state-of-the-art software model checkers.

The paper is organized as follows. In Section 2 we recall some basic definitions on constraint logic programming. In Section 3 we briefly illustrate our software model checking method for proving program safety by presenting a simple example. In Section 4 we describe the syntax of our imperative language and the CLP interpreter that defines its operational semantics. In Section 5 we specify our problem of proving program safety. In Section 6 we describe the overall strategy of iterated specialization, and also some specific strategies for performing the individual specialization steps. In Section 7 we report on the experiments we have performed by using our prototype implementation, and we compare our results with those obtained using ARMC, HSF(C), and TRACER. Finally, in Section 8 we discuss the related work and we compare our approach with other existing methods for performing software model checking.

2. Preliminaries on Constraint Logic Programming

Let us recall some basic notions and terminology concerning constraint logic programming. For more details the reader may refer to [27].

We will consider constraint logic programs with linear constraints over the set $\mathbb{Z}$ of the integer numbers. If $\mathtt{p}_1$ and $\mathtt{p}_2$ are linear polynomials whose variables and coefficients are of type integer then $\mathtt{p}_1 = \mathtt{p}_2$, $\mathtt{p}_1 \geq \mathtt{p}_2$, 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. An *atom* is an atomic formula of the form $\mathtt{p(t_1,\ldots,t_m)}$, where `p` is a predicate symbol not in $\{=, \geq, >\}$ and $\mathtt{t_1,\ldots,t_m}$ are terms. 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. We assume that in every clause all integer arguments in its head are distinct variables. The clause `A :- c` is called a *constrained fact* for the predicate occurring in `A`. If the constraint `c` is `true`, then it is omitted and the constrained fact is called a *fact*. A CLP program is said to be *linear* if all its clauses are of the form `A :- c, B`, where `B` consists of at most one atom.

We say that a predicate `p` *depends on* a predicate `q` in a program P if either in P there is a clause of the form `p(...) :- c, B` such that `q` occurs in `B`, or there exists a predicate `r` such that `p` depends on `r` in P and `r` depends on `q` in P. We say that a predicate `p` in a linear program P is *useless* if in P there are constrained facts neither for `p` nor for each predicate `q` on which `p` depends.

Let $T_{\mathbb{Z}}$ denote the set of ground terms built out of the elements of $\mathbb{Z}$ and the function symbols in the language of the CLP program P. A *$\mathbb{Z}$-interpretation* is an interpretation with universe $T_{\mathbb{Z}}$ such that: (i) it assigns to $+, *, =, \geq, >$ the usual meaning in $\mathbb{Z}$, and (ii) it is the Herbrand interpretation for function and predicate symbols different from $+, *, =, \geq, >$. We can identify a $\mathbb{Z}$-interpretation I with the set of ground atoms (with arguments in $T_{\mathbb{Z}}$) that are true in I.

We write $\mathbb{Z} \models \varphi$ if φ is true in every $\mathbb{Z}$-interpretation. A constraint `c` is *satisfiable* if $\mathbb{Z} \models \exists(\mathtt{c})$, where $\exists(\varphi)$ denotes the existential closure of φ. A constraint is *unsatisfiable* if it is not satisfiable. A constraint `c` *entails* a constraint `d`, denoted $\mathtt{c} \sqsubseteq \mathtt{d}$, if $\mathbb{Z} \models \forall(\mathtt{c} \rightarrow \mathtt{d})$, where $\forall(\varphi)$ denotes the universal closure of φ. We say that a clause of the form `H :- c, B` is *subsumed* by the constrained fact `H :- d` if $\mathtt{c} \sqsubseteq \mathtt{d}$.

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

3. An Introductory Example

In this introductory example we will very briefly present our method for performing software model checking via iterated specialization. Let us consider the following imperative program P:

Program P

```
int x;  int y;  int n;
ℓ0: while (x < n) { x = x + 1;  y = y + 2; }
ℓ1: while (x > 0) { x = x − 1;  y = y − 1; }
ℓh: halt;
```

We want to show safety of this program with respect to the initial configurations satisfying $\varphi_{init}(x,y,n) =_{def} x = 0 \wedge y = 0 \wedge n \geq 0$, and the error configurations satisfying $\varphi_{error}(x,y,n) =_{def} y < n$. That is, we want to show that, starting from any values of x, y, and n that satisfy $\varphi_{init}(x,y,n)$, for every execution of program P, after executing the command ℓ_h: `halt`, the new values of x, y, and n do not satisfy $\varphi_{error}(x,y,n)$.

Our verification method starts off by encoding the unsafety property (that is, the negation of the safety property) as a CLP program I. (i) First we write the CLP clauses defining the predicate `tr` (short, for transition relation) that encodes the interpreter of the imperative language in which the given program P is written. In particular, the predicate `tr` defines the transition relation from any given configuration to the next configuration. (ii) Then, we write the CLP clauses that encode the imperative program P and the formulas φ_{init} and φ_{error}. (iii) Finally, by using the predicate `tr`, we write the CLP clauses that define the predicate `unsafe` that holds iff there exists an execution of program P that leads from an initial configuration to an error configuration. Details of all these encoding steps will be given in the following section.

In order to derive the verification conditions associated with the given program P, we perform a specialization of the CLP program I, called *the removal of the interpreter*, that is, we specialize the clauses defining `unsafe` with respect to: (i) the clauses that define `tr`, and (ii) the clauses that encode P. We get the following CLP program I_{sp}:

```
1. unsafe :- X=0, Y=0, N≥0, new1(X,Y,N).
2. new1(X,Y,N) :- X<N, X1=X+1, Y1=Y+2, new1(X1,Y1,N).
3. new1(X,Y,N) :- X≥N, new2(X1,Y1,N).
4. new2(X,Y,N) :- X>0, X1=X-1, Y1=Y-1, new2(X1,Y1,N).
5. new2(X,Y,N) :- X≤0, Y<N.
```

where `new1` and `new2` are predicates that have been automatically introduced by the specialization algorithm, and correspond to the while-loops at labels ℓ_0 and ℓ_1, respectively. Unfortunately, it is not possible to check by direct evaluation whether or not the atom `unsafe` is a consequence of the above CLP program I_{sp}. Indeed, the evaluation of the query `unsafe` using the standard top-down strategy gets into an infinite loop. Tabled evaluation [11] does not terminate either, as infinitely many tabled atoms are generated. Analogously, bottom-up evaluation is unable to return an answer, because it has to generate infinitely many facts for `new1` and `new2` for deriving that `unsafe` is not a consequence of the given CLP program I_{sp}.

Our verification method avoids the direct evaluation of the above CLP program I_{sp}, and applies some symbolic evaluation methods based on program specialization. Indeed, starting from those clauses, we perform a second specialization, called *the propagation of the constraints*. This second specialization propagates throughout clauses 1–5 the constraint '`X=0, Y=0, N≥0`' characterizing the initial configurations. By doing so, we get the following CLP program I^1_{sp}:

```
6. unsafe :- X=0, Y=0, N≥0, new3(X,Y,N).
7. new3(X,Y,N) :- X1=X+1, 2*X1=Y+2, X1≥1, X1≤N, Y1=2*X1, new3(X1,Y1,N).
8. new3(X,Y,N) :- X=N, Y=2*N, N≥0, new4(X,Y,N).
9. new4(X,Y,N) :- Y-X=N, X1≥0, X1=X-1, Y1=Y-1, Y1>2*X1, new4(X1,Y1,N).
```

where `new3` and `new4` are new predicates introduced by the specialization algorithm. Since in this final program I^1_{sp} there are no constrained facts, its least model is empty. In particular, no atom with predicate `new3` belongs to that model. We conclude that `unsafe` *does not hold* in the least model of program I^1_{sp}. Then, by the correctness of the CLP encoding (see Theorem 1 in Section 5) and by the correctness of CLP program specialization with respect to the least model semantics (see Theorem 2 in Section 6) we get, as desired, that the given imperative program P is safe, that is, no execution of P can lead from an initial configuration to an error configuration.

In the general case it may happen that after the specialization steps we are *not* able to conclude whether or not `unsafe` holds, because the specialized CLP program defines the predicate `unsafe` by recursive clauses with constrained facts. (These clauses may also be mutually recursive.) In this case we do not give up and we use the specialized program I^1_{sp} we have derived as a new initial program for continuing our verification task. Indeed, we iterate program specialization by propagating the constraints characterizing the error configurations and, if required, by propagating again the constraints characterizing the initial configurations, and so on. In this way we may be able to refine our analysis and improve the result of our verification, as shown in Section 7. Clearly, due to undecidability limitations, it may be the case that we never get to a conclusive result. However, as experimentally shown in Section 7, our method turns out to be successful in many significant examples.

4. A CLP Interpreter for an Imperative Language

We assume that the programs to be verified are written in an imperative language, subset of the C intermediate Language [38], manipulating objects of elementary types, such as integers or characters. Here is the syntax of our language.

$x, y, \ldots$	$\in$	*Vars*	(variable identifiers)
$f, g, \ldots$	$\in$	*Functs*	(function identifiers)
$\ell, \ell_1, \ldots$	$\in$	*Labs*	(labels)
const	$\in$	$\mathbb{Z}$	(integer constants, character constants, ...)
type	$\in$	*Types*	(`int`, `char`, ...)
uop, *bop*	$\in$	*Ops*	(unary and binary operators: $+, -, \leq, \ldots$)

prog	::=	decl* fundef* lab_cmd$^+$	(programs)
decl	::=	*type x*	(declarations)
fundef	::=	*type f* (decl*) { decl* lab_cmd$^+$ }	(function definitions)
lab_cmd	::=	ℓ : cmd	(labelled commands)
cmd	::=	x = expr | $x = f$(expr*) | `return` expr |	(commands)
	|	`goto` ℓ | `if (`expr`)` ℓ_1 `else` ℓ_2 | `halt`	
expr	::=	*const* | x | *uop* expr | expr *bop* expr	(expressions)

For reasons of brevity, we will feel free to say ‘command’, instead of ‘labelled command’. As usual, the superscript $^+$ denotes non-empty finite sequences and the superscript * denotes possibly empty finite sequences. We feel free to separate the members of the sequences by colons or semicolons. The `while` commands can be introduced in our language by considering them as abbreviations of suitable sequences of `if-else` and `goto` commands.

We assume that: (i) every label occurs in every program at most once, (ii) the global variables of a program are those introduced in the declarations of the program, (iii) every variable occurrence is either a global occurrence or a local occurrence with respect to any given function definition, and (iv) in every program one may statically determine whether any given variable occurrence is either global or local.

Language restrictions. We assume that in our language: (i) there are no blocks, and thus no nested levels of locality, (ii) the evaluations of expressions and functions have no side effects, (iii) there are no definitions of recursive functions, and (iv) there are neither arrays, nor structures, nor pointers. Despite these restrictions, our language retains its interest because in the class of programs we can write the safety problem is undecidable, and our experiments show that even in this class the techniques implemented in existing tools are not always satisfactory (see Section 7). □

A program P whose declarations are of the form: *type* $z_1; \ldots;$ *type* z_r, is said to be acting on the global variables $z_1, \ldots, z_r$. Without loss of generality, we also assume that the last command of every program is ℓ_h : `halt` and no other `halt` command occurs in any given program.

Now we give the operational semantics of our imperative language. Let us first introduce the following functions and data structures.
(i) A *global environment* $\delta : Vars \rightarrow \mathbb{Z}$, which is a function that maps global variables to their integer values.
(ii) A *local environment* $\sigma : Vars \rightarrow \mathbb{Z}$, which is a function that maps function parameters and local variables to their integer values.
(iii) An *activation frame*, which is a triple of the form $\langle \ell, y, \sigma \rangle$, where: (1) ℓ is the label where to jump after returning from a function call, (2) y is the variable that stores the value returned by a function call, and (3) σ is the local environment to be initialized when making a function call.
(iv) A *configuration*, which is a triple of the form $\langle\!\langle c, \delta, \tau \rangle\!\rangle$, where: (1) c is a labelled command, (2) δ is a global environment, and (3) τ is a list of activation frames. We operate on the list τ of activation frames by the usual *head* (*hd*) and *tail* (*tl*) functions and the right-associative list constructor *cons* (:). The empty list is denoted by [ ]. Given a function identifier f, a variable identifier x, and an integer v, the term *update*(f, x, v) denotes the function f' that is equal to f, except that $f'(x) = v$.

For any program P, for any label ℓ, (i) *at*(ℓ) denotes the command in P with label ℓ, and (ii) *nextlab*(ℓ) denotes the label of the command in P that is written *immediately after* the command with label ℓ. Given a function identifier f, *firstlab*(f) denotes the label of the first command of the definition of the function f in P. For any expression e, any global environment δ, and any local environment σ, $[\![e]\!]\, \delta\, \sigma$ is the integer value of e. For instance, if x is a global variable and $\delta(x) = 5$, then $[\![x+1]\!]\, \delta\, \sigma = 6$.

The operational semantics that defines the interpreter of our imperative language, is given by a binary transition relation between configurations. That relation, denoted $\Longrightarrow$, is defined by the following rules $R1$–$R5$. Obviously, no rule is given for the command `halt`, because no next configuration is generated when `halt` is executed.

($R1$). *Assignment.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma \rangle$ and v be the integer $[\![e]\!]\,\delta\,\sigma$.

If x is a global variable: $\langle\!\langle \ell\!:\!x\!=\!e,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)),\ update(\delta,x,v),\ \tau \rangle\!\rangle$

If x is a local variable: $\langle\!\langle \ell\!:\!x\!=\!e,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(nextlab(\ell)),\ \delta,\ \langle \ell', y, update(\sigma,x,v) \rangle\!:\!tl(\tau) \rangle\!\rangle$

Informally, an assignment updates either the global environment δ or the local environment σ of the topmost activation frame $\langle \ell', y, \sigma \rangle$.

($R2$). *Function call.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma \rangle$. Let $\{x_1, \ldots, x_k\}$ and $\{y_1, \ldots, y_h\}$ be the set of the formal parameters and the set of the local variables, respectively, of the definition of the function f.

$\langle\!\langle \ell\!:\!x\!=\!f(e_1, \ldots, e_k),\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(firstlab(f)),\ \delta,\ \langle nextlab(\ell), x, \overline{\sigma} \rangle\!:\!\tau \rangle\!\rangle$

where $\overline{\sigma}$ is a local environment of the form: $\{\langle x_1, [\![e_1]\!]\,\delta\,\sigma \rangle, \ldots, \langle x_k, [\![e_k]\!]\,\delta\,\sigma \rangle, \langle y_1, n_1 \rangle, \ldots, \langle y_h, n_h \rangle\}$, for some values $n_1, \ldots, n_h$ in $\mathbb{Z}$ (indeed, when the local variables $y_1, \ldots, y_h$ are declared, they are not initialized). Note that since the values of the n_i's are left unspecified, this transition is nondeterministic.

Informally, a function call creates a new activation frame with: (i) the label where to jump after returning from the call, (ii) the variable where to store the returned value, and (iii) the new local environment.

($R3$). *Return.* Let τ be $\langle \ell', y, \sigma \rangle : \langle \ell'', z, \sigma' \rangle : \tau''$ and v be the integer $[\![e]\!]\,\delta\,\sigma$.

If y is a global variable: $\langle\!\langle \ell\!:\!\texttt{return}\ e,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(\ell'),\ update(\delta,y,v),\ tl(\tau) \rangle\!\rangle$

If y is a local variable: $\langle\!\langle \ell\!:\!\texttt{return}\ e,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \delta,\ \langle \ell'', z, update(\sigma',y,v) \rangle\!:\!\tau'' \rangle\!\rangle$

Informally, a `return` command first evaluates the expression e and computes the value v to be returned, then erases the topmost activation frame $\langle \ell', y, \sigma \rangle$, and finally updates either the global environment δ or the local environment σ' of the new topmost activation frame $\langle \ell'', z, \sigma' \rangle$.

($R4$). *Conditional.* Let $hd(\tau)$ be the activation frame $\langle \ell', y, \sigma \rangle$.

If $[\![e]\!]\,\delta\,\sigma = true$: $\langle\!\langle \ell\!:\ \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(\ell_1),\ \delta,\ \tau \rangle\!\rangle$

If $[\![e]\!]\,\delta\,\sigma = false$: $\langle\!\langle \ell\!:\ \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2,\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(\ell_2),\ \delta,\ \tau \rangle\!\rangle$

($R5$). *Jump.* $\langle\!\langle \ell\!:\!goto\ \ell',\ \delta,\ \tau \rangle\!\rangle \implies \langle\!\langle at(\ell'),\ \delta,\ \tau \rangle\!\rangle$

Given a program P acting on the global variables $z_1, \ldots, z_r$, we define an *initial configuration* to be a triple: $\langle\!\langle \ell_0\!:\!c_0,\ \delta_{init},\ [\,] \rangle\!\rangle$, where: (i) $\ell_0 : c_0$ is the first command of P, (ii) δ_{init} is the initial global environment of the form: $\{\langle z_1, n_1 \rangle, \ldots, \langle z_r, n_r \rangle\}$, where $n_1, \ldots, n_r$ are some given integers in $\mathbb{Z}$, and (iii) $[\,]$ is the empty list of activation frames.

The CLP interpreter for our imperative language is given by the following clauses that define the binary predicate `tr` encoding the transition relation $\implies$ between configurations. We have the clauses for: (i) assignments to global and local variables (clause 1), (ii) function calls and returns (clauses 2 and 3), (iii) conditionals (clauses 4 and 5), (iv) jumps (clause 6), and (v) updates of global and local environments (clauses 7 and 8).

```
1. tr(cf(cmd(L,asgn(X,expr(E))),D,T), cf(cmd(L1,C),D1,T1)) :- loc_env(T,S), eval(E,D,S,V),
                        update(D,T,X,V,D1,T1), nextlab(L,L1), at(L1,C).
2. tr(cf(cmd(L,asgn(X,call(F,Es))),D,T), cf(cmd(FL,C),D,[frame(L1,X,FEnv)|T])) :-
                        nextlab(L,L1), loc_env(T,S), eval_list(Es,D,S,Vs),
                        build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C).
3. tr(cf(cmd(L,return(E)),D,[frame(L1,X,S)|T]), cf(cmd(L1,C),D1,T1)) :-
                        eval(E,D,S,V), update(D,T,X,V,D1,T1), at(L1,C).
4. tr(cf(cmd(L,ite(E,L1,L2)),D,T), cf(cmd(L1,C),D,T)) :- loc_env(T,S), beval(E,D,S),
                        at(L1,C).
5. tr(cf(cmd(L,ite(E,L1,L2)),D,T), cf(cmd(L2,C),D,T)) :- loc_env(T,S), beval(not(E),D,S),
                        at(L2,C).
6. tr(cf(cmd(L,goto(L1)),D,T), cf(cmd(L1,C),D,T)) :- at(L1,C).
7. update(D,T,X,V,D1,T) :- global(X), update_global(D,X,V,D1).
8. update(D,T,X,V,D,T1) :- local(X), update_local(T,X,V,T1).
```

The term `asgn(X,expr(E))` encodes the assignment of the value of the expression `E` to the variable `X`. The term `cmd(L,C)` encodes the command `C` with label `L`. The predicate `loc_env(T,S)` extracts the local environment `S` from the topmost activation frame of the list `T` of activation frames. The predicate `eval(E,D,S,V)` computes the value `V` of the expression `E` in the global environment `D` and the local environment `S`. The predicate `eval_list` is the extension to lists of the predicate `eval`. The predicate `beval(E,D,S)` holds if the boolean expression `E` is true in the global environment `D` and the local environment `S`.

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 that is written immediately after the command with label `L`. The term `frame(L,X,S)` encodes the activation frame made out of the label `L`, the variable `X`, and the local environment `S`. The predicate `firstlab(F,L1)` binds to `L1` the label of the first command of the definition of the function `F`. The predicate `build_funenv(F,Vs,FEnv)` builds the new local environment `FEnv` where the body of the function `F` should be executed, by using the list `Vs` of the values of the actual parameters of the function call.

The term `ite(E,L1,L2)` encodes the if-then-else command jumping to either label `L1` or label `L2`, according the boolean value of the expression `E`. The term `goto(L)` encodes the jump to label `L`. The predicates `global(X)` and `local(X)` hold if `X` denotes a global or local variable, respectively. The predicate `update_global(D,X,V,D1)` updates the global environment `D` by binding the variable `X` to the value `V`, thereby constructing a new global environment `D1`. Similarly, the predicate `update_local(T,X,V,T1)` updates the local environment of the topmost activation frame in the list `T`, thereby constructing a new list `T1` of activation frames.

Note that the CLP clauses 1–8 are clauses without constraints in their bodies. However, constraints are used in the definitions (not presented here) of the predicates `eval` and `beval`.

5. The Safety Problem

The problem of verifying the safety of a program P is the problem of checking whether or not, starting from an initial configuration, the execution of P leads to a so-called error configuration. This problem is formalized by defining an *unsafety triple* of the form: $\{\!\{\varphi_{init}(z_1,\ldots,z_r)\}\!\}\ P\ \{\!\{\varphi_{error}(z_1,\ldots,z_r)\}\!\}$, where:

(i) P is a program acting on the global variables $z_1,\ldots,z_r$,

(ii) $\varphi_{init}(z_1,\ldots,z_r)$ is a *disjunction* of constraints that characterizes the values of the global variables in the initial configurations, and

(iii) $\varphi_{error}(z_1,\ldots,z_r)$ is a *disjunction* of constraints that characterizes the values of the global variables in the error configurations.

We say that a program P is *unsafe* with respect to a set of initial configurations satisfying $\varphi_{init}(z_1,\ldots,z_r)$ and a set of error configurations satisfying $\varphi_{error}(z_1,\ldots,z_r)$ or simply, *P is unsafe* with respect to φ_{init} and φ_{error}, if there exist two global environments δ_{init} and δ_h such that the following conjunction holds:

$$\begin{aligned}
&\varphi_{init}(\delta_{init}(z_1),\ldots,\delta_{init}(z_r))\\
\wedge\quad &\langle\!\langle \ell_0\!:\!c_0, \delta_{init}, [\,]\rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h\!:\!\mathtt{halt}, \delta_h, [\,]\rangle\!\rangle\\
\wedge\quad &\varphi_{error}(\delta_h(z_1),\ldots,\delta_h(z_r))
\end{aligned}$$

where, as already mentioned, $\ell_0\!:\!c_0$ is the first command of P and $\ell_h\!:\!\mathtt{halt}$ is the last command of P. As usual, $\Longrightarrow^*$ denotes the reflexive, transitive closure of $\Longrightarrow$.

A program is said to be *safe* with respect to φ_{init} and φ_{error} if it is not unsafe with respect to φ_{init} and φ_{error}.

An unsafety triple can be encoded as a CLP program. We show how this encoding can be done through the following example. The extension to the general case is straightforward.

Let us consider the unsafety triple:

$$\{\!\{\varphi_{init}(x,y,n)\}\!\}\ \ \mathit{Increment}\ \ \{\!\{\varphi_{error}(x,y,n)\}\!\}$$

where: (i) $\varphi_{init}(x,y,n)$ is $x=0 \wedge y=0$, (ii) $\varphi_{error}(x,y,n)$ is $x>y$, and (iii) the program *Increment* acting on the global variables x, y, and n, is:

```
int x;   int y;   int n;                                    Program Increment
ℓ0: while (x<n) { x = x + 1;  y = x + y; }
ℓh: halt;
```

The `while` command stands for the following sequence of labelled commands of our imperative language:

ℓ_0: `if` $(x{<}n)$ ℓ_1 `else` ℓ_h ;
ℓ_1: $x = x + 1$;
ℓ_2: $y = x + y$;
ℓ_3: `goto` ℓ_0 ;

Then, the program *Increment* is encoded by the following clauses:

```
1. at(0,ite(less(int(x),int(n)),1,h)).
2. at(1,asgn(int(x),expr(plus(int(x),int(1))))).
3. at(2,asgn(int(y),expr(plus(int(x),int(y))))).
4. at(3,goto(0)).
5. at(h,halt).
```

We also consider the following CLP clauses that specify the reachability relation from an initial configuration to an error configuration:

$\boxed{\text{CLP Program } P1}$

```
6. unsafe :- initConf(X), reach(X).
7. reach(X) :- tr(X,X1), reach(X1).
8. reach(X) :- errorConf(X).
9. initConf(cf(cmd(0,ite(less(int(x),int(n)),1,h)), [[int(x),X],[int(y),Y],[int(n),N]],[]))
            :- X=0, Y=0.
10. errorConf(cf(cmd(h,halt), [[int(x),X],[int(y),Y],[int(n),N]],[])) :- X>Y.
```

These clauses, together with: (i) the clauses, listed in Section 4 that define the predicate `tr`, and (ii) the clauses defining the predicates on which the predicate `tr` depends (as, for instance, the above clauses 1–5 defining the predicate `at`), constitute a CLP program, which we call $P1$.

We have that clauses 9 and 10 specify the initial and error configurations of the given unsafety triple. In these two clauses the global environment (that is, the second component of the configuration) has been encoded by the list `[[int(x),X],[int(y),Y],[int(n),N]]` that gives the bindings for the program variables x, y, and n, respectively. In the initial configuration (see clause 9) we have the first command of our program, that is, the command `cmd(0,ite(less(int(x),int(n)),1,h))`, and the initial list of activation frames, which is the empty list [ ]. In the error configuration (see clause 10) we have the last command of our program, that is, `cmd(h,halt)`.

The CLP program $P1$ is said to be the *CLP encoding* of the unsafety triple $\{\!\{\varphi_{init}(x,y,n)\}\!\}$ *Increment* $\{\!\{\varphi_{error}(x,y,n)\}\!\}$. (Note that clauses 6, 7, and 8 are common to the CLP encodings of all unsafety triples.)

The following theorem shows that the encoding of the safety problem into CLP clauses is correct.

Theorem 1. (Correctness of CLP Encoding) *Let I be the CLP encoding of any given unsafety triple* $\{\!\{\varphi_{init}\}\!\}$ *P* $\{\!\{\varphi_{error}\}\!\}$. *The program P is safe with respect to* φ_{init} *and* φ_{error} *iff* `unsafe` $\notin M(I)$.

Proof. In the CLP encoding I, the predicate `tr` encodes the transition relation $\Longrightarrow$ associated with the given imperative program P, that is, $I \models$ `tr(cf1,cf2)` iff $cf_1 \Longrightarrow cf_2$, where `cf1` and `cf2` are terms encoding the configurations cf_1 and cf_2, respectively. The predicates `initConf` and `errorConf` encode the initial and the error configurations, respectively, that is, the following Properties (A) and (B) hold.
Property (A): $I \models$ `initConf(init-cf)` iff `init-cf` is the term encoding a configuration of the form $\langle\!\langle \ell_0 : c_0, \delta_{init}, [\,] \rangle\!\rangle$ such that $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds, and
Property (B): $I \models$ `errorConf(error-cf)` iff `error-cf` is the term encoding a configuration of the form $\langle\!\langle \ell_h : \texttt{halt}, \delta_h, [\,] \rangle\!\rangle$ such that $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds.

By clauses 7 and 8 of the CLP encoding I and Property (B), for any configuration cf encoded by the term `cf`, we have that $I \models$ `reach(cf)` iff there exists a configuration cf_h of the form $\langle\!\langle \ell_h : \texttt{halt}, \delta_h, [\,] \rangle\!\rangle$ such that $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds and $cf \Longrightarrow^* cf_h$.

Now, by clause 6 of the CLP encoding I and Property (A), we get that $I \models$ `unsafe` iff there exist configurations cf_0 and cf_h such that the following hold:
(i) cf_0 is of the form $\langle\!\langle \ell_0 : c_0, \delta_{init}, [\,] \rangle\!\rangle$,
(ii) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$,
(iii) $cf_0 \Longrightarrow^* cf_h$,

(iv) cf_h is of the form $\langle\!\langle \ell_h : \mathtt{halt}, \delta_h, [\,] \rangle\!\rangle$, and

(v) $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$.

Thus, by the definition of unsafety, $I \models$ `unsafe` iff P is unsafe with respect to φ_{init} and φ_{error}. The thesis follows from the fact that $I \models$ `unsafe` iff `unsafe` $\in M(I)$ [28]. □

Note that, instead of using clauses 6–8, we could have defined the predicate `unsafe` using the following clauses 6'–8' that, given the transition relation `tr`, define the reachability relation in a backward way, from the error configuration back to the initial configuration:

```
6'. unsafe :- errorConf(X), reach(X).
7'. reach(X1) :- tr(X,X1), reach(X).
8'. reach(X) :- initConf(X).
```

As it will be explained in Section 6.5, clauses 6'–8' are semantically equivalent to clauses 6–8, but they may determine a different behavior of the verification method. Indeed, the unfolding of clauses 6–8 propagates the information of the constraints of the initial configuration towards the error configuration, while the unfolding of clauses 6'–8' propagates in a backward way the information of the constraints of the error configuration towards the initial configuration.

In our verification method we propagate, by program specialization, in an alternate manner both kinds of constraints, those of the initial configuration and those of the error configuration, by applying, between any two specializations, the so-called Reverse Transformation (see Section 6.5) that, indeed, reverts the order of the computation [5] and realizes a transformation similar to the one leading from clauses 6–8 to clauses 6'–8' (and vice versa).

6. The Verification Method

In order to verify whether or not a given imperative program P is unsafe with respect to φ_{init} and φ_{error}, by Theorem 1 we can check whether or not the atom `unsafe` belongs to $M(I)$, where I is the CLP encoding of the unsafety triple $\{\!\{\varphi_{init}\}\!\}\ P\ \{\!\{\varphi_{error}\}\!\}$. However, there is no algorithm that always terminates and performs that check because, as already mentioned, safety is undecidable. Indeed, in general, $M(I)$ is an infinite model and its construction may not terminate, and also the standard top-down evaluation strategy (see, for instance, [27]) may not terminate for the query `unsafe`.

As an alternative to the construction of the least model and to the standard query evaluation strategies, in this paper we introduce our software model checking method. It is based on *iterated specialization* and, indeed, it performs a *sequence* of program specializations. The strategy for constructing this sequence of specializations is presented in Figure 1.

```
Input:  A CLP program $I$ encoding an unsafety triple $\{\!\{\varphi_{init}\}\!\}\ P\ \{\!\{\varphi_{error}\}\!\}$.
Output: if $P$ is safe with respect to $\varphi_{init}$ and $\varphi_{error}$ then 'safe' else 'unsafe'.
Step 1. $Specialize_{Remove}(I, I_{sp})$;                                % REMOVING THE INTERPRETER
Step 2. $Specialize_{Prop}(I_{sp}, I')$;                                 % PROPAGATING CONSTRAINTS
Step 3. if $SafetyTest(I')$ = 'safe'     then return 'safe'               % LIGHTWEIGHT ANALYSIS
        if $SafetyTest(I')$ = 'unsafe'   then return 'unsafe'
        if $SafetyTest(I')$ = 'unknown'  then {$Reverse(I', I_{sp})$ ; goto Step 2}
```

Figure 1: The Iterated Specialization strategy.

The Iterated Specialization strategy takes as input the CLP program I made out of: (i) the CLP facts encoding the imperative program P, (ii) the clauses for the interpreter `tr` that encodes the transition relation $\Longrightarrow$, (iii) the clauses for the predicates `unsafe` and `reach` (see clauses 6–8 of Section 5), (iv) the clauses for `initConf` and `errorConf` encoding the formulas φ_{init} and φ_{error}, respectively.

The Iterated Specialization strategy has the objective of deriving a new CLP program, call it I_{sp}, such that: (i) `unsafe` $\in M(I)$ iff `unsafe` $\in M(I_{sp})$, and (ii) I_{sp} either contains the fact `unsafe` or contains no clauses with head `unsafe`. In the former case the unsafety property encoded by I holds and the given imperative program is unsafe, while in the latter case the unsafety property does not hold and the given imperative program P is safe.

The input program I is first specialized by applying the procedure $\mathit{Specialize}_{Remove}$, which realizes the removal of the interpreter (this step is common to other specialization-based techniques for the verification of imperative programs [12, 40]). In particular, $\mathit{Specialize}_{Remove}$ unfolds away the relation `tr` and introduces new predicate definitions corresponding to (a subset of) the 'program points' of the original imperative program. Thus, at the end of this first specialization we derive a CLP program I_{sp} that defines the predicate `unsafe` without referring to the predicate `tr`, and in this sense we say that this first specialization realizes the removal of the interpreter.

Then, the Iterated Specialization strategy applies the procedure $\mathit{Specialize}_{Prop}$, which propagates the constraints of the initial configuration. As shown in the introductory example of Section 3, the constraints of the initial configuration can be propagated through the program I_{sp} obtained after removing the interpreter, by specializing I_{sp} itself with respect to φ_{init}, thereby deriving a new specialized program I'.

Next, our Iterated Specialization strategy performs a lightweight analysis, called the *SafetyTest*, to check whether or not `unsafe` belongs to $M(I')$, that is, whether or not P is unsafe. In particular, *SafetyTest* checks whether I' can be transformed into an equivalent program T where one of the following conditions three holds: either (i) the fact `unsafe` belongs to T, hence there is a computation leading to an error configuration and the strategy halts reporting '*unsafe*', or (ii) T has no constrained facts, hence no computation leads to an error configuration and the strategy halts reporting '*safe*', or (iii) T contains a clause of the form `unsafe :- G`, where `G` is not the empty goal (thus, neither (i) nor (ii) holds), and hence the strategy proceeds to the subsequent step.

In that subsequent step our strategy propagates the constraints of the error configuration. This is done by: (i) first applying the *Reverse* procedure, which, so to say, inverts the flow of computation by interchanging the roles of the initial configuration and the error configuration, and (ii) then specializing (using again the procedure $\mathit{Specialize}_{Prop}$) the 'reversed' program with respect to φ_{error}.

The strategy iterates the applications of the *Reverse* and the $\mathit{Specialize}_{Prop}$ procedures until hopefully *SafetyTest* succeeds, thereby reporting either '*safe*' or '*unsafe*'. Obviously, due to the undecidability of safety, the Iterated Specialization strategy may not terminate. However, we will show that each iteration terminates, and hence we can refine the analysis and possibly increase the level of precision by starting each new iteration from the CLP program obtained as output of the previous iteration.

6.1. The Specialize Procedure

The $\mathit{Specialize}_{Remove}$ and $\mathit{Specialize}_{Prop}$ procedures are two specific versions of the generic *Specialize* procedure presented in Figure 2.

Let us assume that the CLP program I taken as input by the *Specialize* procedure contains j (≥ 1) clauses that define the predicate `unsafe` and they are of the form:

$\mathtt{unsafe :- c_1(X), A_1(X)}, \quad \ldots, \quad \mathtt{unsafe :- c_j(X), A_j(X)}$

where $\mathtt{c_1(X)}, \ldots, \mathtt{c_j(X)}$ are either atoms or constraints, and $\mathtt{A_1(X)}, \ldots, \mathtt{A_j(X)}$ are atoms. (Here and in the rest of this section by `c(X)` and `A(X)` we denote a constraint or an atom, respectively, whose variables are among those in the tuple `X`.) For instance, if program I is the CLP program $P1$ of Section 5, we have that j is 1, $\mathtt{c_1(X)}$ is the atom `initConf(X)`, and $\mathtt{A_1(X)}$ is the atom `reach(X)`.

The *Specialize* procedure modifies the initial program I by propagating the information present in the constraints $\mathtt{c_1(X)}, \ldots, \mathtt{c_j(X)}$ that characterize the initial or the error configuration. (The fact that they characterize either an initial or an error configuration depends on the number of applications of the *Reverse* procedure.) This propagation of information is realized by unfolding steps. In particular, by these unfoldings we may discover that the atom `unsafe` has a successful derivation, and hence the given imperative program is unsafe. Alternatively, by unfolding we may add constraints that are inconsistent with the ones occurring in the constrained facts (if any), and by folding we may derive mutually recursive predicates with no constrained facts, and hence infer that the given imperative program is safe.

The *Specialize* procedure makes use of the *unfolding*, *clause removal*, *definition introduction*, and *folding* transformation rules [16, 17].

Definition 1 (The 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* $\{\mathtt{K}_i \mathtt{:-} \mathtt{c}_i\mathtt{,B}_i \mid i = 1, \ldots, m\}$ *be the set of the*

```
Input: A CLP program $I$.
Output: A CLP program $I_{sp}$ such that $\texttt{unsafe} \in M(I)$ iff $\texttt{unsafe} \in M(I_{sp})$.
INITIALIZATION:
$I_{sp} := \emptyset$;   $InCls := \{\texttt{unsafe:- c}_1\texttt{(X),A}_1\texttt{(X)}, \ldots, \texttt{unsafe:- c}_j\texttt{(X),A}_j\texttt{(X)}\}$;   $Defs := \emptyset$;

(α) while in $InCls$ there is a clause $C$ that is not a constrained fact do
   UNFOLDING:
      $SpC := Unf(C, \texttt{A}, I)$, where A is the leftmost atom in the body of $C$;
      (α1) while in $SpC$ there is a clause $D$ whose body contains an occurrence of an unfoldable atom A do
               $SpC := (SpC - \{D\}) \cup Unf(D, \texttt{A}, I)$
           end-while;
   CLAUSE REMOVAL:
      (α2) while in $SpC$ there are two distinct clauses $E_1$ and $E_2$ such that $E_1$ subsumes $E_2$ do
               $SpC := SpC - \{E_2\}$
           end-while;
   DEFINITION INTRODUCTION:
      (α3) while in $SpC$ there is a clause $E$ that is not a constrained fact and cannot be folded using a definition
               in $Defs$ do
               $Defs := Defs \cup \{Gen(E, Defs)\}$;   $InCls := InCls \cup \{Gen(E, Defs)\}$;
           end-while;
   $InCls := InCls - \{C\}$;   $I_{sp} := I_{sp} \cup SpC$;
end-while;
FOLDING:
(β) while in $I_{sp}$ there is a clause $E$ that can be folded by a clause $D$ in $Defs$ do
        $I_{sp} := (I_{sp} - \{E\}) \cup \{F\}$, where $F$ is derived by folding $E$ using $D$;
end-while;
Remove from $I_{sp}$ all clauses for predicates on which unsafe does not depend.
```

Figure 2: The *Specialize* Procedure.

(*renamed apart*) *clauses in program I such that, for* $i = 1, \ldots, m$, A *is unifiable with* $\texttt{K}_i$ *via the most general unifier* ϑ_i *and* $(\texttt{c},\texttt{c}_i)\,\vartheta_i$ *is satisfiable. We define the following function Unf*:

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

Each clause in $Unf(C, \texttt{A}, I)$ *is said to be derived by* unfolding C w.r.t. A *using I.*

In order to perform unfolding during specialization, we assume that the atoms occurring in bodies of clauses are annotated as either *unfoldable* or *not unfoldable*. This annotation is based on an analysis of program *I* which ensures that any sequence of clauses constructed by unfolding w.r.t. unfoldable atoms is finite. We will provide more details on the annotations used by the $Specialize_{Remove}$ and $Specialize_{Prop}$ procedures in Sections 6.2 and 6.3, respectively. We refer to [33] for a survey of techniques for controlling unfolding and guaranteeing this finiteness property.

The *Specialize* procedure introduces a set *Defs* of new *predicate definitions*, that is, a set of clauses of the form newp(X) :- c(X), A(X), where newp is a predicate symbol not occurring in *I* and occurring in *Defs* only once. New predicate definitions are introduced by using a function *Gen*, called the *generalization operator*, which we now define. The definition of *Gen* extends to clauses the definition of *widening operator* on constraints considered in the field of program analysis [7].

Definition 2 (The Generalization Operator). *Let E be a clause of the form* H(X) :- e(X,X1), Q(X1), *where* X *and* X1 *are tuples of variables,* e(X,X1) *is a constraint, and* Q(X1) *is an atom. Let Defs be a set of predicate definitions.*

Then, Gen(E, Defs) is a clause G: `newq(X1) :- g(X1), Q(X1)`, *such that*: (i) `newq` *is a new predicate symbol, and* (ii) `e(X,X1)` $\sqsubseteq$ `g(X1)`.

For any infinite sequence $E_1, E_2, \ldots$ *of clauses, let* $G_1, G_2, \ldots$ *be a sequence of clauses constructed as follows*: (1) $G_1 = Gen(E_1, \emptyset)$, *and* (2) *for every* $i > 0$, $G_{i+1} = Gen(E_{i+1}, \{G_1, \ldots, G_i\})$. *We assume that the sequence* $G_1, G_2, \ldots$ stabilizes, *that is, there exists an index k such that, for every* $i > k$, G_i *is equal, modulo the head predicate name, to a clause in* $\{G_1, \ldots, G_k\}$.

In Sections 6.2 and 6.3 we will present some specific generalization operators that are used in our experiments of Section 7. More generalization operators that are used for the specialization of logic programs and constraint logic programs can be found in [17, 21, 33, 34, 39].

Definition 3 (The Folding Rule). *Let Defs be a set of predicate definitions. Let us consider a clause E of the form*: `H(X) :- e(X,X1), Q(X1)`, *and a clause D in Defs of the form*: `newq(X1) :- g(X1), Q(X1)` *such that* `e(X,X1)` $\sqsubseteq$ `g(X1)`. *By* folding *E* using *D we derive the clause F*: `H(X) :- e(X,X1), newq(X1)`.

Termination and Correctness of Specialization

The correctness of the *Specialize* procedure with respect to the least model semantics directly follows from the correctness of the transformation rules that are used [16]. Indeed, the *Specialize* procedure enforces all the applicability conditions for the unfolding and folding rules presented in [16].

As mentioned above, the termination of the unfolding phase of the *Specialize* procedure is guaranteed by a suitable annotation of the atoms in the body of the clauses. Moreover, when *Gen* is a generalization operator defined as indicated in Definition 2, a *finite* set of new predicates are introduced during the *Specialize* procedure, and hence that procedure terminates.

Thus, we have the following result.

Theorem 2. (Termination and Correctness of Specialization) (i) *The Specialize procedure terminates.* (ii) *Let program* I_{sp} *be the output of the Specialize procedure applied to the input program I. Then* `unsafe` $\in M(I)$ *iff* `unsafe` $\in M(I_{sp})$.

Proof. *Point* (i). In order to prove the termination of the *Specialize* procedure we assume that the *unfoldable* and *not unfoldable* annotations guarantee the termination of the unfolding while-loop ($\alpha 1$). Since the clause removal while-loop ($\alpha 2$), the definition introduction while-loop ($\alpha 3$), and the folding while-loop (β) clearly terminate, we are left with the task of showing that the first, outermost while-loop (α) terminates, that is, we have to show that the number of new predicate definitions added to *InCls* by definition introduction is finite. This finiteness is guaranteed by the following facts:
(1) all new predicate definitions are introduced by using the generalization operator *Gen*,
(2) by Definition 2 the set of all new predicate definitions generated by a sequence of applications of *Gen* is finite, modulo the head predicate names, and
(3) no two new predicate definitions that are equal modulo the head predicate name are introduced by definition introduction. Indeed, definition introduction introduces a new predicate definition only if the definitions already present in *Defs* cannot be used to fold clause *E* (see the while-loop ($\alpha 3$)).

Point (ii). Before proving the correctness of the *Specialize* procedure, let us briefly recall the results for CLP transformation rules presented in [16, 18]. We refer to the adaptation of the transformation rules to CLP program specialization considered in [20]. First, we need the following notion.

A *transformation sequence* is a sequence $P_0, \ldots, P_n$ of CLP programs constructed from an initial program P_0 by applying the unfolding, clause removal, definition introduction, and folding rules. By $Defs_i$, with $0 \leq i \leq n$, we denote the set of clauses introduced by the definition introduction rule during the construction of the subsequence $P_0, \ldots, P_i$. In a transformation sequence, program P_{i+1} is derived from P_i by applying one of the transformation rules as follows:
(*Unfolding*) $P_{i+1} = (P_i - \{C\}) \cup Unf(C, \mathtt{A}, P_i)$, where C is a clause in P_i and `A` is an atom in the body of C;
(*Clause Removal*) $P_{i+1} = P_i - \{C\}$, where C is a clause in P_i that is subsumed by a constrained fact in $P_i - \{C\}$;
(*Definition Introduction*) $P_{i+1} = P_i \cup \{$`newp(X) :- c(X), A(X)`$\}$, where `newp` is a predicate symbol not occurring in $P_0, \ldots, P_i$;

(*Folding*) $P_{i+1} = (P_i - \{E\}) \cup \{F\}$, where E is a clause in P_i and F is a clause derived by folding E using a clause D in $Defs_i$.

The following result is proved in [16, 20].

Theorem (Correctness of the Transformation Rules). Let $P_0, \ldots, P_n$ be a transformation sequence. Let us assume that for every i, with $0 < i < n-1$, if P_{i+1} is derived by folding a clause in P_i using a clause D in $Defs_i$, then there exists j, with $0 < j < n-1$, such that: (i) D belongs to P_j, and (ii) P_{j+1} is derived by unfolding D w.r.t. an atom in its body. Then, for every ground atom `A` whose predicate occurs in P_0, we have that `A` $\in M(P_0)$ iff `A` $\in M(P_n)$.

The *Specialize* procedure constructs a transformation sequence $P_0, \ldots, P_n$, such that:
(i) P_0 is I, and
(ii) P_n is $(I - \{$`unsafe:- c`$_1$`(X),A`$_1$`(X)`$, \ldots,$ `unsafe:- c`$_j$`(X),A`$_j$`(X)`$\}) \cup I'$, where I' is the value of I_{sp} at the exit of the Folding while-loop (β). (Note that, for $i = 0, \ldots, n-1$, $Unf(C, \texttt{A}, P_i) = Unf(C, \texttt{A}, P_0)$, for every clause C and atom `A`.)

The hypothesis of the Theorem of Correctness of the Transformation Rules is fulfilled, as all clauses in *Defs* are unfolded. Thus, `unsafe` $\in M(I)$ iff `unsafe` $\in M(P_n)$. The thesis follows from the fact that, by deleting from P_n the clauses defining predicates on which `unsafe` does not depend, we get a final program I_{sp} such that `unsafe` $\in M(I_{sp})$ iff `unsafe` $\in M(P_n)$. □

6.2. *Removal of the interpreter*

As already mentioned, the removal of the interpreter is achieved by applying the $Specialize_{Remove}$ procedure to the program I that encodes an unsafety triple $\{\!\{\varphi_{init}\}\!\}\ P\ \{\!\{\varphi_{error}\}\!\}$. $Specialize_{Remove}$ is a specific version of the *Specialize* procedure where unfoldings and generalizations steps are performed as we will indicate below.

An atom `A` in the body of a clause C is *unfoldable* iff one of the following conditions holds:
(i) the predicate of `A` is different from `reach`,
(ii) `A` is of the form `reach(cf(cmd(lab,c),d,t))` and `c` is either `asgn(x,expr(e))`, or `return(e)`, or `halt`,
(iii) `A` is of the form `reach(cf(cmd(lab,goto(nextlab)),d,t))` and C has *not* been derived (in one or more steps) by unfolding a clause with respect to an atom of the form `reach(cf(cmd(lab,goto(nextlab)),d',t'))`.

Note that a `reach` atom containing a command of the form `asgn(x,call(...))` is assumed to be not unfoldable. Condition (iii) allows unfolding with respect to a `reach` atom containing a `goto` command, but prevents infinite unfolding. Finally, note that a `reach` atom containing an `ite` command is not unfoldable, and hence a potential exponential blow-up due to the unfolding of conditionals is avoided. Indeed, it can easily be shown that the size of the output program I_{sp} of $Specialize_{Remove}$ is linear with respect to the size of input program I (and thus, also with respect to the size of the imperative program P of the triple encoded by I).

The generalization operator *Gen* used in the $Specialize_{Remove}$ procedure is defined as follows.

Given a clause E: `H(X) :- e(X,X1), Q(X1)` and a set *Defs* of predicate definitions, $Gen(E, Defs)$ is the clause `newq(X1) :- Q(X1)`, where `newq` is a new predicate symbol. (With reference to Definition 2, the constraint `g(X1)` is `true`.) This generalization operator guarantees that, as required by Definition 2, the set of new predicate definitions introduced by using *Gen* is finite. Indeed, `Q(X1)` must be of the form `reach(cf(cmd(lab,c),d,t))` and there is a finite number of distinct quadruples (`lab,c,d,t`), as there is a finite number of distinct (global or local) variable identifiers and, under our assumption that P does not contain recursive function definitions, the lists of activation frames generated by unfolding have bounded length.

Let us consider again the unsafety triple $\{\!\{x = 0 \wedge y = 0)\}\!\}$ *Increment* $\{\!\{x > y\}\!\}$ of our running example presented in Section 5. The first step of the *Iterated Specialization* strategy consists in applying the $Specialize_{Remove}$ strategy to the CLP program $P1$ encoding the given unsafety triple.

We start off by executing the first while-loop (α), which iterates the Unfolding, Clause Removal, and Definition Introduction steps.

First execution of the body of the while-loop (α).
Unfolding. By unfolding clause 6 with respect to `initConf(X)` (which is the leftmost atom in the body of clause 6), we get:

```
11. unsafe :- X=0, Y=0, reach(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
                 [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

No Clause Removal is applicable.

Definition Introduction. The `reach` atom in clause 11 is not unfoldable because it contains an `ite` command. Thus, we stop unfolding and we introduce the new predicate definition:

```
12. new1(X,Y,N) :- reach(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
                   [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

Second execution of the body of the while-loop (α).
Unfolding. Then we unfold clause 12 and we get the two clauses:

```
13. new1(X,Y,N) :- tr(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
                   [[int(x),X],[int(y),Y],[int(n),N]],[]),X1), reach(X1).
14. new1(X,Y,N) :- errorConf(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
                   [[int(x),X],[int(y),Y],[int(n),N]],[])).
```

The `tr` and `errorConf` atoms in the bodies of clauses 13 and 14, respectively, are unfoldable. Thus, we perform some more unfolding steps, which can be viewed as mimicking the symbolic evaluation of the `if-else` command, and from clause 13 we get the following two clauses:

```
15. new1(X,Y,N) :- X<N, reach(cf(cmd(1,asgn(int(x),expr(plus(int(x),int(1))))),
                   [[int(x),X],[int(y),Y],[int(n),N]],[])).
16. new1(X,Y,N) :- X≥N, reach(cf(cmd(h,halt),[[int(x),X],[int(y),Y],[int(n),N]],[])).
```

(Note that the test on the condition `less(int(x),int(n))` in the command of clause 13 generates the two constraints `X<N` and `X≥N`.) Then, we delete clause 14 because, by unfolding it, we do not get any clause (indeed, the term `cmd(0,...)` does not unify with the term `cmd(h,...)`).

The `reach` atom in the body of clause 15 is unfoldable, because it contains a command of the form `asgn(int(x), expr(...))`. From clause 15, after two unfolding steps, we get:

```
17. new1(X,Y,N) :- X<N, tr(cf(cmd(1,asgn(int(x),expr(plus(int(x),int(1))))),
                   [[int(x),X],[int(y),Y],[int(n),N]],[]),X1), reach(X1).
```

Also the `reach` atom in the body of clause 16 is unfoldable. After two unfolding steps, we get:

```
18. new1(X,Y,N) :- X≥N, errorConf(cf(cmd(h,halt),[[int(x),X],[int(y),Y],[int(n),N]],[])).
```

Then, by unfolding clause 17 with respect to the unfoldable atom `tr(...)` in its body, we get:

```
19. new1(X,Y,N) :- X<N, X1=X+1, reach(cf(cmd(2,asgn(int(y),expr(plus(int(x),int(y))))),
                   [[int(x),X1],[int(y),Y],[int(n),N]],[])).
```

Then, by unfolding clause 18 w.r.t. the atom `errorConf(...)`, we get:

```
20. new1(X,Y,N) :- X≥N, X>Y.
```

The `reach` atom in the body of clause 19 is unfoldable. After two unfolding steps, we get:

```
21. new1(X,Y,N) :- X<N, X1=X+1, tr(cf(cmd(1,asgn(int(y),expr(plus(int(x),int(y))))),
                   [[int(x),X1],[int(y),Y],[int(n),N]][]),X2), reach(X2).
```

By unfolding clause 21 we get:

```
22. new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, reach(cf(cmd(3,goto(0)),
                   [[int(x),X1],[int(y),Y1],[int(n),N]],[])).
```

The sequence of clauses 11, 15, 19, and 22, which we have obtained by unfolding, mimics the execution of the sequence of the four commands: (i) ℓ_0 : `if` $(x < n)$ ℓ_1 `else` ℓ_h, (ii) $\ell_1 : x = x+1$, (iii) $\ell_2 : y = x+y$, and (iv) ℓ_3 : `goto` ℓ_0 (note in those clauses the atoms `reach(cf(cmd(`$i, \ldots$`),`$\ldots, \ldots$`))`, for $i = 0, 1, 2, 3$). Indeed, in general, by unfolding one can perform the symbolic execution of the commands of any given program. The conditions that should hold so that a particular command `cmd(`$i, \ldots$`)` is executed, are given by the constraints in the clause where the atom `reach(cf(cmd(`$i, \ldots$`),`$\ldots, \ldots$`))` occurs.

The `reach` atom in the body of clause 22 is unfoldable, because clause 22 has not been derived from another clause containing a `goto` command. From clause 22, after some more unfolding steps, we get:

```
23. new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, reach(cf(cmd(0,ite(less(int(x),int(n)),1,h)),
                   [[int(x),X1],[int(y),Y1],[int(n),N]],[])).
```

The `reach` atom in the body of clause 23 is not unfoldable, because it contains the `ite` command. However, clause 23 can be folded using definition 12. Thus, the while-loop (α) terminates without introducing any new definition.

The program I_{sp} we have derived so far is made out of clauses 11, 20, and 23 (see Figure 3, where an arrow from n to m denotes that clause m has been derived from clause n by unfolding).

$$\begin{tikzcd}
& \cancel{14} & & & & & & \\
12 \arrow[ur] \arrow[dr] & & 16 \arrow[r] & 18 \arrow[r] & \underline{20} & & & \\
& 13 \arrow[ur] \arrow[dr] & & & & & & \\
& & 15 \arrow[r] & 17 \arrow[r] & 19 \arrow[r] & 21 \arrow[r] & 22 \arrow[r] & \underline{23}
\end{tikzcd}$$

Figure 3: The sequence of unfolding steps starting from clause 12. Clause 14 is deleted by unfolding.

Now, we proceed to the Folding phase. By folding clauses 11 and 23 using definition 12, we get the following final, specialized program $P2$:

```
11.f unsafe :- X=0, Y=0, new1(X,Y,N).                                CLP Program P2
23.f new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
20.  new1(X,Y,N) :- X≥N, X>Y.
```

Note that the folding of clause 23 using the definition for predicate `new1` has been possible because the execution of the program goes back to the `ite` command to which the definition of `new1` refers.

6.3. Propagation of constraints

The procedure $Specialize_{Prop}$ specializes the CLP program I_{sp}, obtained after the application of $Specialize_{Remove}$, by propagating the constraints that characterize the initial or the error configuration. (In our running example program I_{sp} is program $P2$.) Now we describe how the unfolding and generalization steps are performed during $Specialize_{Prop}$.

In order to guide the application of the unfolding rule during the application of $Specialize_{Prop}$, we stipulate that every atom with a new predicate introduced during a previous application of the *Specialize* procedure is annotated as *not unfoldable*. Thus, we can unfold a clause with respect to an atom with a new predicate only once at the first step of the Unfolding phase. This choice guarantees the termination of the Unfolding phase (notice that new predicates can depend on themselves), and also avoids an undesirable, excessive increase of the number of clauses in the specialized program. Obviously, more sophisticated strategies for guiding unfolding could be applied. For instance, one may allow unfolding only if it generates at most one clause. This unfolding policy, called *determinate unfolding* in [22], does not increase the number of clauses and is useful in most cases. However, as we will show in Section 7, the simple choice we have made is effective in practice.

The generalization operator adopted during the $Specialize_{Remove}$ is not adequate for $Specialize_{Prop}$, as it generalizes every constraint to `true`, thereby losing all information about the constraints that characterize the initial and the error configurations. In order for $Specialize_{Prop}$ to be effective, we need to use a generalization operator that retains suitable constraints, while guaranteeing the termination of the specialization.

Now we will consider four generalization operators and in Section 7 we will compare them with respect to their strength for the task of verifying program properties. These generalization operators are based on the *widening* and *convex hull* operators which have been proposed for the static analysis of programs [7, 10] and also applied to the specialization of constraint logic programs (see, for instance, [21, 39]). These generalization operators have been extensively studied in the above cited papers.

Let us briefly recall the notions that are needed to understand the specific operators we have used in our experiments whose results have been reported in Section 7.

Let $\mathbb{R}$ denote the set (and the structure) of the real numbers. We now define the projection, the widening, and the convex hull operators on the reals $\mathbb{R}$. Those operators on the reals have more efficient implementations than those on the integers $\mathbb{Z}$ (where, moreover, projection is not always defined). As shown in [20] the use of operators defined on reals for specializing CLP programs on integers preserves correctness, although it may, at least in principle, decrease precision. Let $\sqsubseteq_{\mathbb{R}}$ denote the entailment relation on the reals $\mathbb{R}$. Clearly, if $\mathtt{c} \sqsubseteq_{\mathbb{R}} \mathtt{d}$ then $\mathtt{c} \sqsubseteq \mathtt{d}$.

The *projection* of a constraint `c(X,Y)` onto the tuple `Y` of variables is a constraint $\texttt{c}_p$`(Y)` such that $\mathbb{R} \models \forall$`Y`($\texttt{c}_p$`(Y)` $\leftrightarrow$ $\exists$`X c(X,Y)`). We have that `c(X,Y)` $\sqsubseteq_{\mathbb{R}}$ $\texttt{c}_p$`(Y)`, and hence `c(X,Y)` $\sqsubseteq$ $\texttt{c}_p$`(Y)`.

Let `c` be a constraint that we assume to have rewritten as a conjunction of inequalities (if `c` is not already in this form). The *widening* of `c` with respect to the constraint `d` is the conjunction of all atomic constraints of `c` that are entailed by `d` [10]. If `w` is the widening of `c` with respect to `d`, then `c` $\sqsubseteq$ `w` and `d` $\sqsubseteq$ `w`.

The *convex hull* of two constraints `c` and `d` is the least (w.r.t. the $\sqsubseteq_{\mathbb{R}}$ ordering) constraint `h` such that `c` $\sqsubseteq_{\mathbb{R}}$ `h` and `d` $\sqsubseteq_{\mathbb{R}}$ `h`. Thus, we also have that `c` $\sqsubseteq$ `h` and `d` $\sqsubseteq$ `h`. The convex hull operator is often used for improving the precision of widening-based static analysis and specialization [10, 39].

We say that a definition `newp(X) :- c(X), Q(X)` is *more general* than a definition `newq(X) :- d(X), Q(X)` if `d(X)` $\sqsubseteq$ `c(X)`.

Let us first present two *monovariant* generalization operators each of which, during the construction of *Defs*, for any given atom `Q(X)` to be folded, introduces a new definition of the form `newp(X) :- c(X), Q(X)`, which is more general than any other definition in *Defs* with the atom `Q(X)` in its body. Thus, when using these generalization operators, the definitions in *Defs* whose body contains the atom `Q(X)` are linearly ordered with respect to the 'more general' relation.

• *Monovariant Generalization with Widening.* This operator is denoted Gen_M. Let E be a clause of the form `H(X) :- e(X,X1), Q(X1)` and *Defs* be a set of predicate definitions. Then,

(μ1) if in *Defs* there is no clause in whose body the atom is (a variant of) `Q(X1)`, then $Gen_M(E, Defs)$ is defined as the clause `newp(X1) :-` $\texttt{e}_p$`(X1), Q(X1)`, and

(μ2) if in *Defs* there is a definition D of the form `newq(X1) :- d(X1), Q(X1)` and D is the most general such definition in *Defs*, then $Gen_M(E, Defs)$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to $\texttt{e}_p$`(X1)`.

Note that at any given time the last definition of the form: `newp(X1) :- c(X1), Q(X1)` that has been introduced in *Defs* is the most general one having the atom `Q(X1)` in its body.

• *Monovariant Generalization with Widening and Convex Hull.* This operator, denoted Gen_{MH}, alternates the computation of convex hull and widening. Formally, $Gen_{MH}(E, Defs)$ is defined like $Gen_M(E, Defs)$, except that in Case (μ2) above, if D has been derived by projection or widening, then $Gen_{MH}(E, Defs)$ is `newp(X1) :- ch(X1), Q(X1)`, where `ch(X1)` is the convex hull of `d(X1)` and $\texttt{e}_p$`(X1)`.

Other generalization operators can be defined by computing any fixed number of consecutive convex hulls before applying widening.

Now we present two *polyvariant* generalization operators, which may introduce several distinct, specialized definitions (with different constraints) for each atom. Polyvariant operators allow, in principle, more precision with respect to monovariant operators, but they may cause the introduction of too many new predicates, and hence an increase of both the size of the specialized program and the time needed for verification. We will consider this issue in Section 7 when we discuss the experiments we have performed.

For these polyvariant generalization operators, we have to consider *Defs* as a *tree* of definitions (rather than simply a *set* of definitions), where definition D is a *child* of definition C if D is introduced to fold a clause derived by unfolding C. We define the *ancestor* relation on *Defs* as the reflexive, transitive closure of the child relation.

When we use a polyvariant generalization operator, for any given atom `Q(X)`, the definitions in *Defs* whose bodies contain `Q(X)` are not necessarily linearly ordered with respect to the 'more general' relation. However, the definitions whose bodies contain `Q(X)` and belong to the same path of the definition tree *Defs* are linearly ordered, and the last definition introduced is more general than all its ancestors.

• *Polyvariant Generalization with Widening.* This operator is denoted Gen_P. Let E be a clause of the form `H(X) :- e(X,X1), Q(X1)` and *Defs* be a tree of predicate definitions. Suppose that E has been derived by unfolding a definition C. Then,

(π1) if in *Defs* there is no ancestor of C in whose body the atom is (a variant of) `Q(X1)`, then $Gen_P(E, Defs)$ is the clause `newp(X1) :-` $\texttt{e}_p$`(X1), Q(X1)`, and

(π2) if C has a most recent ancestor D in *Defs* of the form `newq(X1) :- d(X1), Q(X1)`, then $Gen_P(E, Defs)$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to $\texttt{e}_p$`(X1)`.

• *Polyvariant Generalization with Widening and Convex Hull.* This operator, denoted Gen_{PH} alternates the computation of convex hull and widening. Formally, $Gen_{PH}(E, Defs)$ is defined like $Gen_{P}(E, Defs)$, except that Case (π2) above is modified as follows:

($\pi 2_{PH}$) if C has a most recent ancestor D in *Defs* of the form `newq(X1) :- d(X1), Q(X1)` that has been derived by projection or widening, then $Gen_{PH}(E, Defs)$ is the clause `newp(X1) :- ch(X1), Q(X1)`, where `ch(X1)` is the convex hull of `d(X1)` and `e`$_p$`(X1)`, else $Gen_{PH}(E, Defs)$ is the clause `newp(X1) :- w(X1), Q(X1)`, where `w(X1)` is the widening of `d(X1)` with respect to `e`$_p$`(X1)`.

Now we show that all four operators introduced above are indeed generalization operators in the sense of Definition 2, and hence they guarantee the correctness and termination of $Specialize_{Prop}$. The proof of this fact is similar to the one presented in [21] for similar generalization operators. (Note, however, that in [21] the CLP program given as input to the specialization procedure encodes the satisfaction relation for the Computational Tree Logic.)

Proposition 1. *The operators Gen_M, Gen_{MH}, Gen_P, and Gen_{PH} are generalization operators.*

Proof. By Definition 2, we have to show that, for each operator *Gen* in the set {Gen_M, Gen_{MH}, Gen_P, and Gen_{PH}}, the following two properties hold.
Property (P1): for every clause E of the form: `H(X) :- e(X,X1), Q(X1)`, for every clause `newp(X1) :- g(X1), Q(X1)` obtained by applying the operator *Gen* to E and some set *Defs* of definitions, we have that `e(X,X1)` $\sqsubseteq$ `g(X1)`, and
Property (P2): for every infinite sequence $E_1, E_2, \ldots$ of clauses, for every infinite sequence $G_1, G_2, \ldots$ of clauses constructed as follows: (1) $G_1 = Gen(E_1, \emptyset)$, and (2) for every $i > 0$, $G_{i+1} = Gen(E_{i+1}, \{G_1, \ldots, G_i\})$, there exists an index k such that, for every $i > k$, G_i is equal, modulo the head predicate name, to a clause in $\{G_1, \ldots, G_k\}$.

(Gen_M is a generalization operator)
– Let us prove that Property (P1) holds for Gen_M. If `g(X1)` is `e`$_p$`(X1)` (see case (μ1) above), then `e(X,X1)` $\sqsubseteq$ `e`$_p$`(X1)` because `e`$_p$`(X1)` is the projection of `e(X,X1)` onto `X1`. If `g(X1)` is `w(X1)` (see case (μ2) above), then `e(X,X1)` $\sqsubseteq$ `w(X1)` because `w(X1)` is the widening of `d(X1)` with respect to `e`$_p$`(X1)`, and hence `e`$_p$`(X1)` $\sqsubseteq$ `w(X1)`.
– Let us prove that Property (P2) holds for Gen_M. This property is a straightforward consequence of the following two facts.
(i) Each new definition introduced by Gen_M is a clause of the form `newp(X1) :- g(X1), Q(X1)`, where `Q(X1)` is a function-free atom whose predicate symbol occurs in the input program I. (Indeed, `Q(X1)` is function-free because the input program I is generated by *either* the procedure $Specialize_{Remove}$ that removes, by folding, all function symbols occurring in the atoms of its input program, *or* the procedure $Specialize_{Prop}$ that does not introduce any function symbol.) Thus, we have that Case (μ1) can occur a finite number of times only.
(ii) If `g(X1)` is the widening of `d(X1)` with respect to `e`$_p$`(X1)`, and `g(X1)` is different from `d(X1)`, then the set of atomic constraints of `g(X1)` is a proper subset of the atomic constraints of `d(X1)`, and hence Case (μ2) will eventually generate new predicate definitions whose body is equal to the body of previously generated definitions.

(Gen_{MH} is a generalization operator)
The proof is similar to that for Gen_M.
– In order to prove that Property (P1) holds for Gen_{MH}, we use the fact that `e(X,X1)` $\sqsubseteq$ `ch(X1)`. Indeed, `e(X,X1)` $\sqsubseteq$ `e`$_p$`(X1)`, and `e`$_p$`(X1)` $\sqsubseteq$ `ch(X1)` (because `ch(X1)` is the convex hull of `d(X1)` and `e`$_p$`(X1)`).
– In order to show that Property (P2) holds for Gen_{MH}, it is enough to note that Property (P2) is preserved if one interleaves the projection operator and widening operator with an application of the convex hull operator, and hence the proof already done for Gen_M readily extends to Gen_{MH}.

(Gen_P and Gen_{PH} are generalization operators)
The proof is a straightforward extension of the proof for Gen_M and Gen_{MH}, respectively. In particular, in order to show that Property (P2) holds for Gen_P and Gen_{PH}, it suffices to use the following fact. Suppose that $G_1, G_2, \ldots$ is an infinite sequence of predicate definitions. Let T be an infinite *tree* of definitions such that:
(i) if G occurs in $G_1, G_2, \ldots$, then G occurs in T,
(ii) if A is an ancestor of B in T, then A precedes B in $G_1, G_2, \ldots$ (that is, $G_1, G_2, \ldots$ is a linear order consistent with the ancestor relation in T),

(iii) T is finitely branching, and
(iv) every branch in T stabilizes.
Then $G_1, G_2, \ldots$ stabilizes. □

Different strategies can be adopted for applying the folding rule during the Folding phase. These folding strategies depend on the generalization operator that is used for introducing new definitions. In the case where we use a monovariant operator (either Gen_M or Gen_{MH}), we fold every clause of the form `H(X) :- e(X,X1), Q(X1)` using the most general definition of the form `newq(X1) :- g(X1), Q(X1)` occurring in the set *Defs* obtained at the end of the execution of the *while-loop* (α). We call this strategy the *most general folding strategy*. In the case where we use a polyvariant operator (that is, Gen_P or Gen_{PH}), we fold each clause E using the definition computed by applying the generalization operator to E. We call this strategy the *immediate folding strategy*.

Now let us continue the presentation of our running example which refers to the unsafety triple $\{\!\{x=0 \wedge y=0)\}\!\}$ *Increment* $\{\!\{x>y\}\!\}$. We perform our second program specialization starting from the CLP program $P2$, consisting of the clauses 11.f, 23.f, and 20 we have derived by removing the interpreter. This second specialization propagates the constraint '`X=0, Y=0`' characterizing the initial configuration that occurs in clause 11.f.

We apply the $Specialize_{Prop}$ strategy with the Gen_P generalization operator. We start off by executing the *while-loop* (α) that repeats the Unfolding, Clause Removal, and Definition Introduction steps.

First execution of the body of the while-loop (α).
Unfolding. We unfold clause 11.f with respect to the atom `new1(X,Y,N)` and we get:

24. `unsafe :- X1=1, Y1=1, N>0, new1(X1,Y1,N).`

(Note that `new1(X,Y,N)` is also unifiable with the head of clause 20, but the constraint '`X=0,Y=0,X≥N,X>Y`' is unsatisfiable.) No Clause Removal can be applied.

Definition Introduction. Clause 24 cannot be folded, and hence we define the following new predicate:

25. `new2(X,Y,N) :- X=1, Y=1, N>0, new1(X,Y,N).`

Thus, the set *Defs* of new predicate definitions consists of clause 25 only.

Second execution of the body of the while-loop (α).
Unfolding. Now we unfold the last definition which has been introduced, that is, clause 25, and we get:

26. `new2(X,Y,N) :- X=1, Y=1, X1=2, Y1=3, N>1, new1(X1,Y1,N).`

Definition Introduction. Clause 26 cannot be folded by using any definition in *Defs*. Thus, we apply the generalization operator Gen_P. This operator matches the constraint appearing in the body of clause 26 against the constraint appearing in the body of clause 25, which is the only clause in *Defs*. First, the constraint in clause 25 is rewritten as a conjunction `c` of inequalities: `X≥1,X≤1,Y≥1,Y≤1,N>0`. Then the variables of clause 26 are renamed so that the atom in its body is identical to the atom in the body of clause 25, as follows:

27. `new2(Xr,Yr,Nr) :- Xr=1, Yr=1, X=2, Y=3, N>1, new1(X,Y,N).`

Then the generalization operator Gen_P computes the projection of the constraint appearing in clause 27 onto the variables of the atom `new1(X,Y,N)`, which is the constraint `d`: `X=2,Y=3,N>1`. The widening of `c` with respect to `d` is the constraint '`X≥1,Y≥1,N>0`' obtained by taking the atomic constraints of `c` that are entailed by `d`. Thus, Gen_P introduces the following new predicate definition:

28. `new3(X,Y,N) :- X≥1, Y≥1, N>0, new1(X,Y,N).`

which is added to *Defs*.

Third execution of the body of the while-loop (α).
Unfolding. We unfold clause 28 and we get:

29. `new3(X,Y,N) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).`
30. `new3(X,Y,N) :- X≥N, X>Y, Y≥1, N>0.`

Clause 29 can be folded using clause 28 in *Defs*. Thus, the while-loop (α) terminates without introducing any new definition.

Folding. Now we fold clause 24 using definition 25 and clauses 26 and 29 using definition 28. We get the following final program $P3$:

CLP Program $P3$

```
24.f unsafe :- X1=1, Y1=1, N>0, new2(X1,Y1,N).
26.f new2(X,Y,N) :- X=1, Y=1, N>1, X1=2, Y1=3, new3(X1,Y1,N).
29.f new3(X,Y,N) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y, new3(X1,Y1,N).
30.  new3(X,Y,N) :- X≥N, X>Y, Y≥1, N>0.
```

The application of *Specialize*$_{Prop}$ has propagated the constraints defining the initial configuration. For instance, the constrained fact of $P3$ (see clause 30) has the constraint 'X≥N, X>Y, Y≥1, N>0', while the constrained fact in $P2$ has the constraint 'X≥N, X>Y' only (see clause 20). However, in program $P3$ the presence of a constrained fact does not allow us to conclude that $P3$ has an empty least model, and hence at this point we are not yet able to show the safety of our program *Increment*.

6.4. Lightweight Safety Analysis

The procedure *SafetyTest* (see Figure 4) analyzes the CLP program I' derived by specializing I_{sp} (in our running example program I' is program $P3$, and program I_{sp} is program $P2$), and tries to determine whether or not unsafe belongs to $M(I')$.

The analysis performed by *SafetyTest* is lightweight in the sense that, unlike the *Iterated Specialization* strategy, it always terminates, possibly returning the answer '*unknown*'.

Note that the output program I' of *Specialize*$_{Prop}$ is a linear program. The *SafetyTest* procedure transforms I' into a new, linear CLP program T by using two auxiliary functions: (1) the *UnfoldCfacts* function, which takes a linear CLP program T_1 and applies as long as possible the following transformation: $T_1 := (T_1 - \{C\}) \cup \mathit{Unf}(C, \mathtt{A}, T_1)$, where C is any clause in T_1 and A is an atom in the body of C whose definition consists of a set of constrained facts, and (2) the *Remove* function, which takes a linear CLP program T_2 and removes every clause C that satisfies one of the following two conditions: *either* (i) the head predicate of C is useless in T_2, *or* (ii) C is subsumed by a clause in T_2 distinct from C.

The *SafetyTest* procedure iterates the application of the function *UnfoldCfacts* followed by *Remove* until a fixpoint, say T, is reached.

Now we prove that *SafetyTest* constructs the fixpoint program T in a finite number of steps. Moreover, T is equivalent to I' with respect to the least model semantics, and hence unsafe $\in M(I')$ iff unsafe $\in M(T)$.

Theorem 3 (Termination and Correctness of *SafetyTest*). *Let I' be a linear CLP program defining the predicate* unsafe. *Then,*
(i) *SafetyTest terminates for the input program I' returning the output 'safe' or 'unsafe' or 'unknown', and*
(ii) *If SafetyTest(I') = 'safe', then* unsafe $\notin M(I')$.
(iii) *If SafetyTest(I') = 'unsafe', then* unsafe $\in M(I')$.

Proof. (i) *SafetyTest* constructs a fixpoint of the function $\lambda T.\mathit{Remove}(\mathit{UnfoldCfacts}(T))$ in a finite number of steps, as we now show. For a CLP program P, let $pn(P)$ denote the number of distinct predicate symbols that occur in the body of a clause in P, and let $cn(P)$ denote the number of clauses in P. Then, the following facts hold:
(1) either $P = \mathit{UnfoldCfacts}(P)$ or $pn(P) > pn(\mathit{UnfoldCfacts}(P))$,
(2) $pn(P) \geq pn(\mathit{Remove}(P))$,
(3) either $P = \mathit{Remove}(P)$ or $cn(P) > cn(\mathit{Remove}(P))$.
Thus, $\langle pn(P), cn(P)\rangle \geq_{lex} \langle pn(\mathit{Remove}(\mathit{UnfoldCfacts}(P))), cn(\mathit{Remove}(\mathit{UnfoldCfacts}(P)))\rangle$, where $\geq_{lex}$ is the lexicographic ordering on pairs of integers. Since $>_{lex}$ is well-founded, *SafetyTest* eventually generates a program T such that $\langle pn(T), cn(T)\rangle = \langle pn(\mathit{Remove}(\mathit{UnfoldCfacts}(T))), cn(\mathit{Remove}(\mathit{UnfoldCfacts}(T)))\rangle$, and hence, by (1) and (3), $T = \mathit{Remove}(\mathit{UnfoldCfacts}(T))$.
(ii) No application of the folding rule is performed by *SafetyTest*, and hence the condition for the correctness of the transformation rules mentioned in the proof of Theorem 2 is trivially satisfied. Thus, unsafe $\in M(I')$ iff unsafe $\in M(T)$. If *SafetyTest*(I') = '*safe*', then no clause in T has head predicate unsafe, and then unsafe $\notin M(T)$. Hence, unsafe $\notin M(I')$.
(iii) If *SafetyTest*(I') = '*unsafe*', then the fact unsafe belongs to T, and unsafe $\in M(T)$. Since at Point (ii) we have shown that unsafe $\in M(I')$ iff unsafe $\in M(T)$, we conclude that unsafe $\in M(I')$. □

Input: A linear CLP program I' defining the predicate `unsafe`.
Output: Either '*safe*' (implying `unsafe` $\notin M(I')$), or '*unsafe*' (implying `unsafe` $\in M(I')$), or '*unknown*'.

```
T := I';
while  T ≠ Remove(UnfoldCfacts(T)) do
        T := Remove(UnfoldCfacts(T));
end-while;

if the fact unsafe belongs to T  then  'unsafe'
else if no clause in T has head predicate  unsafe  then  'safe'  else  'unknown'
```

Figure 4: The *SafetyTest* Procedure.

In our running example, program $P3$ is the CLP program obtained by applying the procedure $Specialize_{Prop}$. Now program $P3$ consists of clauses 24.f, 26.f, 29.f, and 30 and *SafetyTest*($P3$) returns '*unknown*'. Indeed, (i) in program $P3$ no predicate is defined by constrained facts only, and hence *UnfoldCfacts* has no effect, (ii) in $P3$ no predicate is useless and no clause is subsumed by any other, and hence also *Remove* leaves $P3$ unchanged, and (iii) in $P3$ there is a clause for `unsafe` which is not a fact.

6.5. The Reverse Transformation

The *Reverse* procedure implements a transformation that reverses the flow of computation: the top-down evaluation (that is, the evaluation from the head to the body of a clause) of the transformed program corresponds to the bottom-up evaluation (that is, the evaluation from the body to the head) of the given program. In particular, if the *Reverse* procedure is applied to a program that checks the reachability of the error configurations by exploring the transitions in a forward way starting from the initial configurations, then the transformed program checks reachability of the initial configurations by exploring the transitions in a backward way starting from the error configurations. Symmetrically, from a program that checks reachability by a backward exploration of the transitions, *Reverse* derives a program that checks reachability by a forward exploration of the transitions.

Let us consider a linear CLP program Q of the form:

```
unsafe :- a_1(X), p_1(X).
    ...
unsafe :- a_k(X), p_k(X).
q_1(X) :- t_1(X,X1), r_1(X1).
    ...
q_m(X) :- t_m(X,X1), r_m(X1).
s_1(X) :- b_1(X).
    ...
s_n(X) :- b_n(X).
```

where: (i) $a_1(X), \ldots, a_k(X), t_1(X,X1), \ldots, t_m(X,X1), b_1(X), \ldots, b_n(X)$ are constraints, and (ii) $p_1, \ldots, p_k$, $q_1, \ldots, q_m$, $r_1, \ldots, r_m$, and $s_1, \ldots, s_n$ are possibly non-distinct predicate symbols.

The *Reverse* procedure transforms program Q in two steps as follows.

Step 1. Program Q is transformed into a program S of the following form (round parentheses make a single argument out of a tuple of arguments so that, for instance, (p_1,X) is a single argument of the predicate `a`):

```
s1. unsafe :- a(U), r1(U).
s2. r1(U) :- trans(U,V), r1(V).
s3. r1(U) :- b(U).
    a((p_1,X)) :- a_1(X).
     ...
    a((p_k,X)) :- a_k(X).
    trans((q_1,X),(r_1,X1)) :- t_1(X,X1).
     ...
    trans((q_m,X),(r_m,X1)) :- t_m(X,X1).
```

```
b((s1,X)) :- b1(X).
 ...
b((sn,X)) :- bn(X).
```

Step 2. Then program S is transformed into a program Q_{rev} by replacing the first three clauses s1–s3 of S by the following ones:

```
r1. unsafe :- b(U), r2(U).
r2. r2(V) :- trans(U,V), r2(U).
r3. r2(U) :- a(U).
```

The correctness of the transformation of Q into Q_{rev} is shown by the following result.

Theorem 4. *Let Q_{rev} be the program derived from program Q by the Reverse procedure. Then* `unsafe` $\in M(Q)$ *iff* `unsafe` $\in M(Q_{rev})$.

Proof. (*Step* 1.) By unfolding clauses s1, s2, and s3 of program S with respect to `a(U)`, `trans(U,V)`, and `b(U)`, respectively, we get the following CLP program Q':

```
unsafe :- a1(X), r1((p1,X)).
   ...
unsafe :- ak(X), r1((pk,X)).
r1((q1,X)) :- t1(X,X1), r1((r1,X1)).
   ...
r1((qm,X)) :- tm(X,X1), r1((rm,X1)).
r1((s1,X)) :- b1(X).
   ...
r1((sn,X)) :- bn(X).
```

By the correctness of the unfolding rule (see the theorem stating the correctness of the transformation rules in the proof of Theorem 2), we get that `unsafe` $\in M(S)$ iff `unsafe` $\in M(Q')$.

Then, by rewriting all atoms in Q' of the form `r1((pred,Z))` into `pred(Z)`, we get back Q. (The occurrences of predicate symbols in the arguments of `a`, `trans`, and `b` should be considered as individual constants.) The correctness of this rewriting is straightforward, as it is based on the syntactic isomorphism between Q and Q'. A formal proof of correctness can be made by observing that the above rewriting can also realized by introducing predicate definitions of the form `pred(Z) :- r1((pred,Z))`, and then applying the unfolding and folding rules. Thus, `unsafe` $\in M(Q')$ iff `unsafe` $\in M(Q)$.

(*Step* 2.) The transformation of S into Q_{rev} (and the opposite transformation) can be viewed as a special case of the *grammar-related transformation* studied in [5]. We refer to that paper for a proof of correctness. Thus, we have that `unsafe` $\in M(S)$ iff `unsafe` $\in M(Q_{rev})$, and we get the thesis. □

The predicates `a`, `trans`, and `b` are assumed to be unfoldable in the subsequent application of the $Specialize_{Prop}$ procedure.

Now let us continue our running example.

The program $P3$ derived by the $Specialize_{Prop}$ procedure at the end of Section 6.3 can be transformed, as indicated at Steps 1 and 2 of the *Reverse* procedure, into the following CLP program $P3_{rev}$:

CLP Program $P3_{rev}$

```
r1. unsafe :- b(U), r2(U).
r2. r2(V) :- trans(U,V), r2(U).
r3. r2(U) :- a(U).
s4. a((new2,X1,Y1,N)) :- N>0, X1=1, Y1=1.
s5. trans((new2,X,Y,N),(new3,X1,Y1,N)) :- X=1, Y=1, N>1, X1=2, Y1=3.
s6. trans((new3,X,Y,N),(new3,X1,Y1,N)) :- X≥1, Y≥1, X<N, X1=X+1, Y1=X1+Y.
s7. b((new3,X,Y,N)) :- Y≥1, N>0, X≥N, X>Y.
```

The idea behind program reversal is best understood by considering the reachability relation in the (possibly infinite) transition graph whose transitions are defined by the (instances of) clauses s5 and s6. The program S consisting of clauses s1–s7 obtained at the end of Step 1, checks the reachability of a configuration $c2$ satisfying `b(U)` from a configuration $c1$ satisfying `a(U)`, by moving *forward* from $c1$ to $c2$. Program $P3_{rev}$, obtained from S by replacing s1–s3 with r1–r3, checks the reachability of $c2$ from $c1$, by moving *backward* from $c2$ to $c1$. Thus, in the case where `a(U)` and `b(U)` are predicates that characterize the initial and error configurations, respectively, by the reversal transformation we derive a program that checks the reachability of the initial configurations by moving *backward* from the error configurations. In particular, in the body of the clause for `unsafe` in $P3_{rev}$ the constraint `b(U)` contains, among others, the constraint `X>Y` characterizing the error configuration (see clause s7) and, by specializing $P3_{rev}$, we will propagate the constraint of the error configuration.

Now let us specialize the CLP program $P3_{rev}$ by applying again *Specialize*$_{Prop}$. We start off from the first while-loop (α) of that procedure.

First execution of the body of the while-loop (α).
Unfolding. We unfold the clause for `unsafe` (that is, clause r1) with respect to the leftmost atom `b(U)` and we get:

```
31. unsafe :- Y≥1, N>0, X≥N, X>Y, r2((new3,X,Y,N)).
```

Definition Introduction. In order to fold clause 31 we introduce the definition:

```
32. new4(X,Y,N) :- Y≥1, N>0, X≥N, X>Y, r2((new3,X,Y,N)).
```

Second execution of the body of the while-loop (α).
Unfolding. Then, we unfold clause 32 and we get:

```
33. new4(X,Y,N) :- Y≥1, N>0, X≥N, X>Y, a((new3,X,Y,N)).
34. new4(X1,Y1,N) :- Y1≥1, N>0, X1≥N, X1>Y1, trans(U,(new3,X1,Y1,N)), r2(U).
```

By unfolding, clause 33 is deleted because the head of clause s4 is not unifiable with `a((new3,X,Y,N))`. By unfolding clause 34 with respect to `trans(U,(new3,X1,Y1,N))`, also this clause is deleted because unsatisfiable constraints are derived. Thus, no new definition is introduced, and the while-loop (α) terminates.

Folding. By folding clause 31 using definition 32 we get:

```
31.f unsafe :- Y≥1, N>0, X≥N, X>Y, new4(X,Y,N).                CLP Program P4
```

and the final, specialized CLP program $P4$ consists of clause 31.f only.

Now we apply the *SafetyTest* procedure to program $P4$. This procedure detects that `unsafe` is a useless predicate and returns the answer '*safe*'. Then, the *Iterated Specialization* terminates by reporting that the given imperative program *Increment* is safe.

Thus, in this example we have seen that by iterating the specializations which propagate the constraints occurring in the initial configuration and in the error configuration, we have been able to show safety of the given program. The effectiveness of iterated specialization is confirmed by the experiments we have performed on various examples taken from the literature, as reported in Section 7.

6.6. Soundness of the Software Model Checking method

Finally, we get the following soundness result for the Iterated Specialization method.

Theorem 5. (Soundness of the Software Model Checking method) *Let I be the CLP encoding of the unsafety triple* $\{\!\{\varphi_{init}\}\!\}$ *P* $\{\!\{\varphi_{error}\}\!\}$. *If the Iterated Specialization strategy terminates for the input program I and returns 'safe', then P is safe with respect to* φ_{init} *and* φ_{error}. *If the strategy terminates and returns 'unsafe', then P is unsafe with respect to* φ_{init} *and* φ_{error}.

Proof. The Iterated Specialization strategy terminates for the input program I and returns '*safe*' (respectively, '*unsafe*') if and only if there exists n such that:

Specialize$_{Remove}(I, I_{sp})$; *Specialize*$_{Prop}(I_{sp}, I^1)$;
Reverse(I^1, I^1_{sp}); *Specialize*$_{Prop}(I^1_{sp}, I^2)$;
. . .
Reverse(I^{n-1}, I^{n-1}_{sp}); *Specialize*$_{Prop}(I^{n-1}_{sp}, I^n)$;

and *SafetyTest*(I^n) = '*safe*' (respectively, *SafetyTest*(I^n) = '*unsafe*').

Then, by Theorem 3,
    `unsafe` $\notin M(I^n)$ (respectively, `unsafe` $\in M(I^n)$)
if and only if (by Theorems 2 and 4)
    `unsafe` $\notin M(I)$ (respectively, `unsafe` $\in M(I)$)
if and only if (by Theorem 1)
    P is safe (respectively, unsafe) with respect to φ_{init} and φ_{error}. □

7. Experimental Evaluation

We have performed an experimental evaluation of our software model checking method on several benchmark programs taken from the literature. The results of our experiments show that our approach is competitive with state-of-the-art software model checkers.

The benchmark set used in our experiments consists of 216 safety verification problems of C programs (179 of which are safe, and the remaining 37 are unsafe). Most problems have been taken from the benchmark sets of other tools used in software model checking, like DAGGER [25] (21 problems), TRACER [30] (66 problems) and InvGen [26] (68 problems), and from the TACAS 2013 Software Verification Competition [2] (52 problems). The size of the input programs ranges from a dozen to about five hundred lines of code.

The verification problems came in different source formats (and, in particular, they used different methods for specifying the initial and error configurations), and thus they could not be directly used by other software model checkers. We automatically converted all problems from their original format to a common, intermediate format, and then from the intermediate format to the format accepted by each tool we have considered in our experiments. We have put great care and effort in writing these conversion programs to ensure maximum compatibility. Nonetheless, some software model checkers failed to run on some seemingly harmless verification problems. The source code of all the verification problems we have considered and detailed reports about the verification results are available at `http://map.uniroma2.it/VeriMAP/scp/`.

We have realized the VeriMAP software model checker [15] that implements our verification method. VeriMAP is based on MAP, a system for the transformation of constraint logic programs, that is written in SICStus Prolog and operates on constraints over the rationals by using the `clpqr` library.

Our software model checker consists of three modules.
(i) A front-end module, *C2CLP*, based on the C Intermediate Language (CIL) and its associated tools [38], which encodes a C program together with the initial and error configurations, into a set of CLP facts. These facts, together with the clauses for the predicates `tr`, `unsafe`, and `reach` (and the predicates they depend upon), are used during the first program specialization which removes the interpreter.
(ii) A module for CLP program transformation, that indeed removes the interpreter and applies the Iterated Specialization strategy.
(iii) A module that implements the *SafetyTest* procedure.

Our software model checker VeriMAP has been configured to execute the following program transformation sequence:

$$Specialize_{Remove};\quad Specialize_{Prop};\quad SafetyTest;\quad (Reverse;\ Specialize_{Prop};\ SafetyTest)^*$$

where, as usual, a^* denotes a finite sequence of a's. Thus, VeriMAP executes: (i) a first program specialization consisting of an application of the $Specialize_{Remove}$ procedure that performs the *removal of the interpreter*, and (ii) a sequence of applications of the $Specialize_{Prop}$ procedure (from now on called *iterations*) that performs the propagation of the constraints of the initial and the error configurations. After the removal of the interpreter, the first application of $Specialize_{Prop}$ propagates the constraints of the initial configuration. This corresponds to a *forward* propagation along the graph of configurations associated with the reachability relation `tr` (see Section 4), while the propagation of the constraints of the error configuration corresponds to a *backward* propagation along the graph of configurations. The $Specialize_{Prop}$ procedure has been executed by using the four generalization operators presented in Section 6: (i) Gen_M, which is a monovariant generalization with widening only, (ii) Gen_{MH}, which is a monovariant generalization with widening and convex hull, (iii) Gen_P, which is a polyvariant generalization with widening only, and (iv) Gen_{PH}, which is polyvariant generalization with widening and convex hull.

We have also tested the following three state-of-the-art CLP-based software model checkers for C programs: (i) ARMC [41], (ii) HSF(C) [23], and (iii) TRACER [30]. ARMC and HSF(C) are based on the Counter-Example Guided Abstraction Refinement technique (CEGAR) [6, 32, 44], while TRACER uses a technique based on approximated preconditions and approximated postconditions. We have compared the performance of those software model checkers with the performance of VeriMAP on our benchmark programs.

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system Ubuntu 12.10 (64 bit) (kernel version 3.2.0-27). A timeout limit of five minutes has been set for all model checkers.

In Table 1 we summarize the verification results obtained by the four software verification tools we have considered. In the column labelled by VeriMAP (Gen_{PH}) we have reported the results obtained by using the VeriMAP system that implements our Iterated Specialization method with the generalization operator Gen_{PH}. In the remaining columns we have reported the results obtained, respectively, by ARMC, HSF(C), and TRACER using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options.

		VeriMAP (Gen_{PH})	ARMC	HSF(C)	TRACER	
					SPost	*WPre*
1	*correct answers*	185	138	159	91	103
2	safe problems	154	112	137	74	85
3	unsafe problems	31	26	22	17	18
4	*incorrect answers*	0	9	5	13	14
5	false alarms	0	8	3	13	14
6	missed bugs	0	1	2	0	0
7	*errors*	0	18	0	20	22
8	*timed-out problems*	31	51	52	92	77
9	*total score*	339 (0)	210 (-40)	268 (-28)	113 (-52)	132 (-56)
10	*total time*	10717.34	15788.21	15770.33	27757.46	23259.19
11	*average time*	57.93	114.41	99.18	305.03	225.82

Table 1: Verification results using VeriMAP, ARMC, HSF(C) and TRACER. For each column the sum of the values of lines 1, 4, 7, and 8 is 216, which is the total number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.

Line 1 reports the total number of correct answers of which those for safe problems and unsafe problems are indicated in line 2 and 3, respectively. Line 4 reports the number of verification tasks that ended with an incorrect answer. These verification tasks refer to safe programs that have been proved unsafe (*false alarms*, at line 5), and unsafe programs that have been proved safe (*missed bugs*, at line 6). Line 7 reports the number of verification tasks that aborted due to some errors (because of insufficient memory or inability of parsing). Line 8 reports the number of verification tasks that did not provide an answer within the timeout limit of five minutes.

The total score obtained by each tool using the score function of the TACAS 2013 Software Verification Competition [2], is reported at line 9 and will be used in Figure 5. The score function assigns to every program p the integer $score(p)$ determined as follows: (i) 2, if p is safe and has been correctly verified, (ii) 1, if p is unsafe and has been correctly verified, (iii) -4, if a false alarm has been generated, and (iv) -8, if a bug has been missed. Programs that caused errors or timed out do not contribute to the total score.

At line 9 we have indicated between round parentheses the negative component of the score due to false alarms and missed bugs. Line 10 reports the total CPU time, in seconds, taken to run the whole set of verification tasks: it includes the time taken to produce the (correct or incorrect) answers and the time spent on tasks that timed out (we did not include the negligible time taken for tasks that aborted due to errors). Line 11 reports the average time needed to produce a correct answer, which is obtained by dividing the total time (line 10) by the number of correct answers (line 1).

On the set of verification problems we have considered, the VeriMAP system provided correct answers to 185 problems out of 216. It is followed by HSF(C) (159 correct answers), ARMC (138), TRACER(*WPre*) (103), and TRACER(*SPost*) (91). Moreover, VeriMAP produced no errors and no incorrect answers, while the other tools generate from 5 to 14 incorrect answers. Thus, VeriMAP exhibits the best *precision*, which is defined as the ratio between the number of correct answers and the number of verification problems.

The total time taken by VeriMAP is smaller than the time taken by any of the other tools we have considered. This result is somewhat surprising, if one considers the generality of our approach and the fact that our system has not been specifically optimized for software model checking. This good performance of VeriMAP is due to: (i) the small number of tasks that timed out, and (ii) the fact that VeriMAP takes very little time on most programs, while it takes much more time on a few, complex programs (depicted in Figure 5). In particular, VeriMAP is able to produce 169 answers taking at most 5 seconds each, and this is indeed a good performance if we compare it with HSF(C) (154 answers), ARMC (122), TRACER(*SPost*) (88) and TRACER(*WPre*) (101).

In order to ease the comparison of the performance of the software model checkers we have considered, we adopt a visualization technique using score-based quantile functions (see Figure 5), similar to that used by the TACAS 2013 Software Verification competition [2]. By using this technique, the performance of each tool is represented by a quantile function, which is a set of pairs (x, y) computed from the following set of pairs:

$V = \{ (p, t) \mid$ the correct answer for the (safe or unsafe) program p is produced in t seconds $\}$.

Given the set V, for each pair $(p, T) \in V$ we produce a pair (x, y) computed as follows:

(i) $x = XS + \sum_{(p,t)\in V \wedge t\leq T} score(p)$, where XS, called the *x-shift*, is the sum of all the negative scores due to incorrect answers (x is called the *accumulated score*), and

(ii) $y = YS + T$, where YS, called the *y-shift*, is a number of seconds equal to the number of timed-out problems.

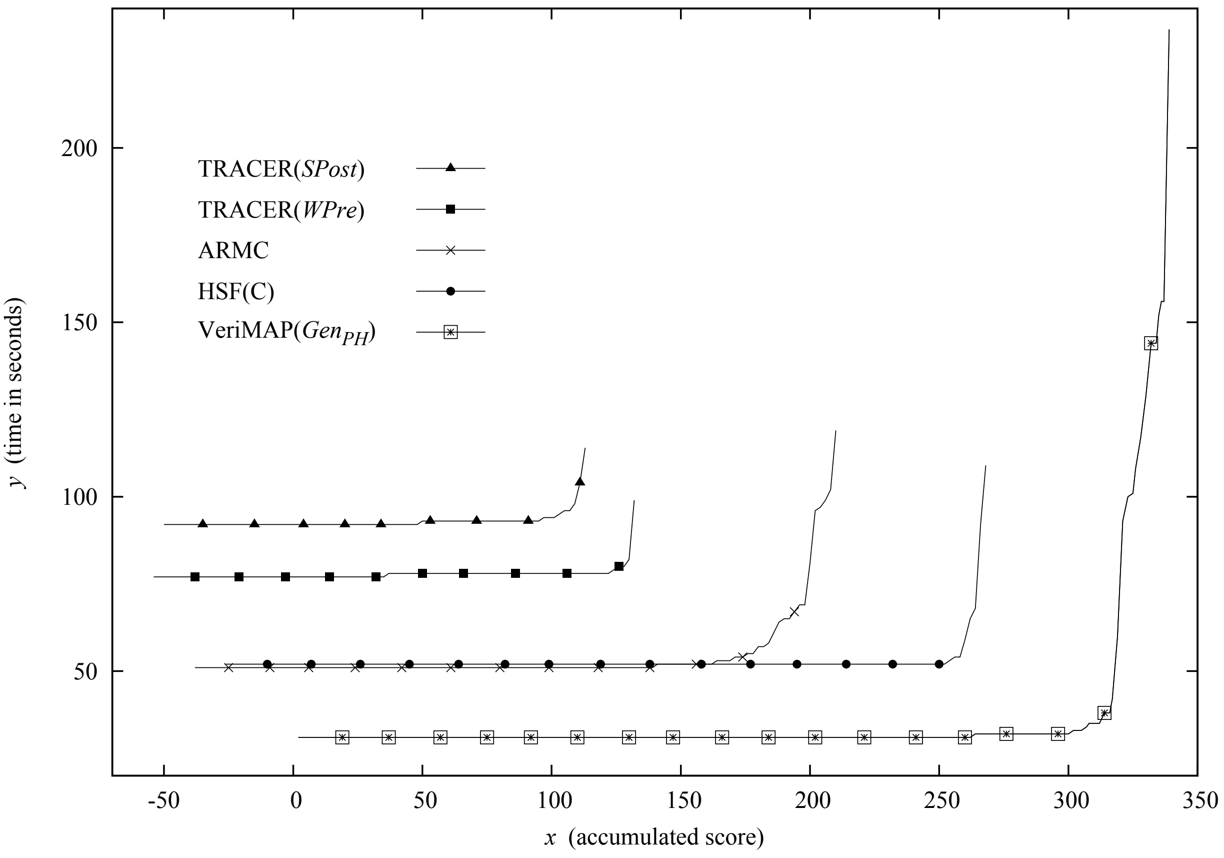

Figure 5: Score-based quantile functions for TRACER(*SPost*), TRACER(*WPre*), ARMC, HSF(C), and VeriMAP(Gen_{PH}). Markers are placed along the lines of the functions, from left to right, every 10 programs that produce correct answers, starting from the program whose verification took the minimum time.

Quantile functions are discrete monotone functions, but for reasons of simplicity, in Figure 5 we depicted them as

continuous lines. The line of a quantile function for a generic verification tool should be interpreted as follows:
(i) the x coordinate of the k-th leftmost point of the line, is the sum of the score of the fastest k correctly verified programs plus the (non-positive) *x-shift*, and measures the reliability of the tool,
(ii) the y coordinate of the k-th leftmost point of the line, is the verification time taken for the k-th fastest correctly verified program plus the (non-negative) *y-shift*, and measures the inability of the tool of providing an answer (either a correct or an incorrect one),
(iii) the span of the line along the x-axis, called the *x-width*, measures the precision of the tool, and
(iv) the span of the line along the y-axis, called the *y-width*, is the difference of verification time between the slowest and the fastest correctly verified program.

Moreover, along each line of the quantile functions of Figure 5 we have put a marker every ten answers. In particular, for $n \geq 1$, the n-th marker, on the left-to–right order, denotes the point (x, y) corresponding to the $(10\,n)$-th fastest correct answer that has been produced.

Informally, for the line of a quantile function we have that: good results move the line to the right (because of lower total negative score XS), stretch it horizontally (because of higher positive accumulated score), move it towards the x-axis (because of fewer timed-out problems and a better time performance), and compress it vertically (because of a lower difference between worst-case and best-case time performance). We observe that the line of the quantile function for VeriMAP(Gen_{PH}) starts with a positive x-value (indeed, VeriMAP provides no incorrect answers and $XS = 0$), is the widest one (indeed, VeriMAP has the highest positive accumulated score, due to its highest precision among the tools we have considered), and is the lowest one (indeed, VeriMAP(Gen_{PH}) has the smallest numbers of timed-out problems). The height of the line for VeriMAP(Gen_{PH}) increases only towards the right end (at an accumulated score value of 300) after having produced 162 correct answers, and this number is greater than the number of all the correct answers produced by any of the other tools we have considered.

In Table 2 we report the results obtained by running the VeriMAP system with the four generalization operators presented in Section 6.

Each column is labelled by the name of the associated generalization operator. Line 1 reports the total number of correct answers. Lines 2 and 3 report the number of correct answers for safe and unsafe problems, respectively. Line 4 reports the number of verification tasks that timed out. As already mentioned, the VeriMAP system has produced neither incorrect answers nor errors. Line 5 reports the total time, including the time spent on tasks that timed out, taken to run the whole set of verification tasks. Line 6 reports the average time needed to produce a correct answer, that is, the total time (line 5) divided by the number of correct answers (line 1). Line 7 reports the total correct answer time, that is the total time taken for producing the correct answers, excluding the time spent on tasks that timed out. Lines 7.1–7.4 report the percentages of the total correct answer time taken to run the *C2CLP* module, the $Specialize_{Remove}$ procedure, the $Specialize_{Prop}$ procedure, and the *SafetyTest* procedure, respectively. Line 8 reports the average correct answer time, that is, the total correct answer time (line 7) divided by the number of correct answers (line 1). Line 9 reports the maximum number of iterations of the Iterated Specialization strategy that were needed, after the removal of the interpreter, for verifying the safety property of interest on the individual programs. Line 10 reports the total number of predicate definitions introduced by the $Specialize_{Prop}$ procedure during the various iterations.

The data presented in Table 2 show that polyvariance always gives better precision than monovariance. Indeed, the polyvariant generalization operator with convex hull Gen_{PH} achieves the best precision (it provides the correct answer for 85.65% of 216 programs of our benchmark), followed by the polyvariant generalization operator without convex hull Gen_P (73.61%). (For this reason in Table 1 we have compared the other verification systems against VeriMAP (Gen_{PH}).) As already mentioned, polyvariant generalization may introduce more than one definition for each program point, which means that the *Specialize* procedure yields a more precise abstraction of the program to be verified and, consequently, it may increase the effectiveness of the safety test. The increase of precision obtained by using polyvariant operators rather than monovariant ones is particularly evident when proving unsafe programs (the precision is increased of about 100%, from 15 to 29–31).

On the other hand, monovariant operators enjoy the best trade-off between precision and average correct answer time. For example, when considering the average correct answer time, the Gen_M operator, despite the significant loss of precision (about 20%, from 159 to 128) with respect to its polyvariant counterpart Gen_P, is about 2.4 times faster (3.25 seconds versus 7.88 seconds), when we consider the average correct answer time.

The good performance of monovariant operators is also justified by the much smaller number of definitions (about

		VeriMAP generalization operators			
		Gen_M	Gen_{MH}	Gen_P	Gen_{PH}
1	*correct answers*	128	147	159	185
2	safe problems	113	132	130	154
3	unsafe problems	15	15	29	31
4	*timed-out problems*	88	69	57	31
5	*total time*	26816.64	21362.93	18353.11	10717.34
6	*average time*	209.51	145.33	115.43	57.93
7	*total correct answer time*	416.64	662.93	1253.11	1417.34
7.1	*C2CLP*	2.27%	1.63%	0.96%	0.98%
7.2	$Specialize_{Remove}$	6.81%	4.74%	4.44%	4.06%
7.3	$Specialize_{Prop}$	90.33%	93.16%	42.77%	44.68%
7.4	*SafetyTest*	0.59%	0.46%	51.83%	50.27%
8	*average correct answer time*	3.25	4.51	7.88	7.66
9	*max number of iterations*	4	7	10	7
10	*number of definitions*	5623	6248	54977	58226

Table 2: Verification results using the VeriMAP system with different generalization operators. The sum of the values of lines 1 and 4 is 216, which is the number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.

one tenth) introduced by the $Specialize_{Prop}$ procedure with respect to those introduced in the case of polyvariant operators.

Also the size of the programs produced by monovariant operators is much smaller with respect to those produced by polyvariant operators. (The size of the final programs obtained by specialization is not reported in Table 2, but it is approximately proportional to the number of definitions.) This also explains why the impact on the total correct answer time of the *SafetyTest* analysis is much lower for monovariant operators (less than 1%) than for polyvariant operators (about 50%).

The weight of the two preliminary phases (translation from C into CLP and removal of the interpreter) on the overall verification process is very limited. The execution times for the *C2CLP* module are very low (about 50 milliseconds per program) and have a low impact on the total correct answer time (at most 2.27%, as reported on line 7.1). The time taken by $Specialize_{Remove}$ for removing the interpreter ranges from a few tenths of milliseconds to about four seconds, for the most complex programs, and its impact on the total correct answer time is between 4% and 7% (see line 7.2).

In the set of problems we have considered, the higher average correct answer time of polyvariant operators does not prevent them from being more precise than monovariant operators. Indeed, by using polyvariant operators we get fewer timed-out problems with respect to those obtained by using monovariant operators, and thus for the verifications that use polyvariant operators we also get smaller total times and average times (which take into account the number of timed-out problems).

We also observe that generalization operators using convex hull always give greater precision than their counterparts that do not use convex hull. This confirms the effectiveness of the convex hull operator, which may help infer relations among program variables, and may ease the discovery of useful program invariants.

Some of the programs are verified by the VeriMAP system during the first iteration after the removal of the interpreter by propagating the constraints of the initial configuration only. Nonetheless, as indicated in Figure 6, which shows the precision achieved by the VeriMAP generalization operators during the first ten iterations, our approach of iterating program specialization is effective and determines a significant increase of the number of correct answers (for Gen_P that number increases from 74 to 159). Moreover, we have that the verification process is able to prove programs by performing up to 7 or 10 iterations when using the more precise generalization operators Gen_{PH} or Gen_P, respectively.

The highest increase of precision is given by the second iteration and, although most correct answers are provided

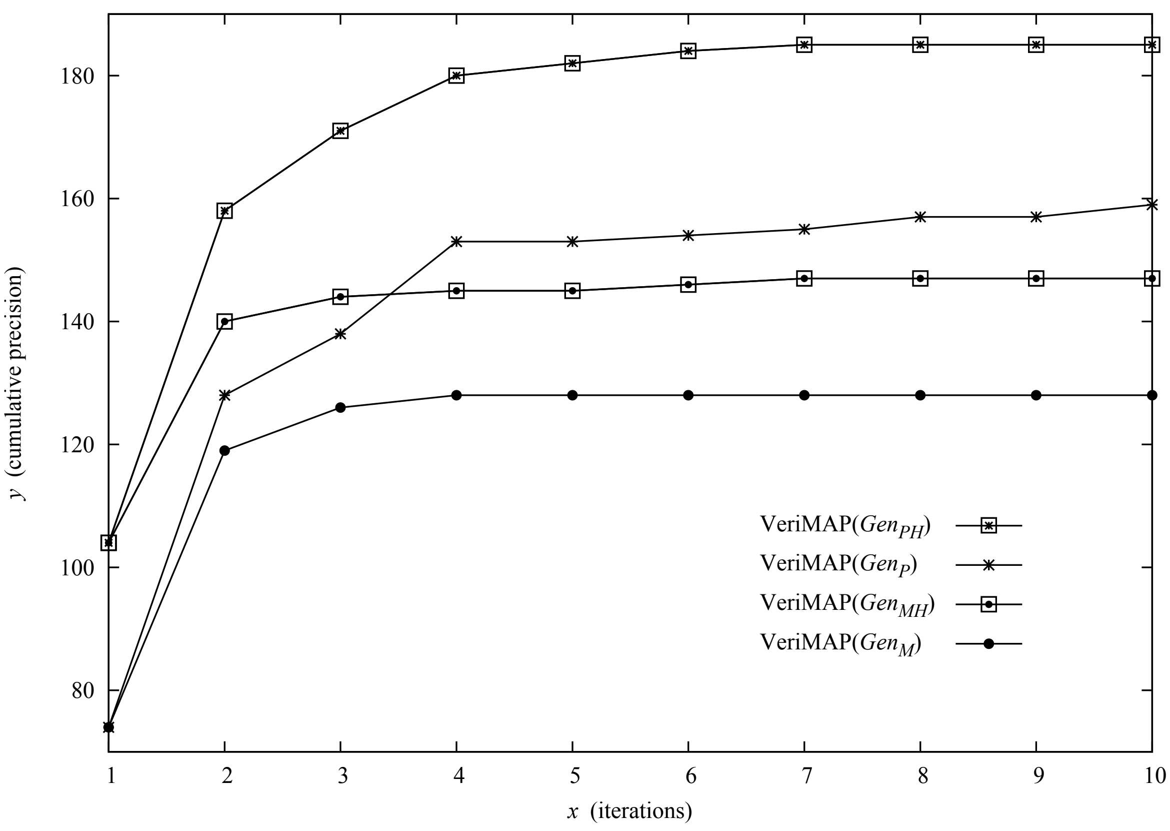

Figure 6: Cumulative precision achieved by the VeriMAP generalization operators during the first ten iterations.

within the fourth iteration, all generalization operators (except for the monovariant operator Gen_M) keep increasing their precision from the fifth iteration onwards, providing up to 6 additional correct answers each.

The iteration of specialization is more beneficial when using polyvariant generalization operators where the increase of the number of answers reaches 115%, while the increase for monovariant generalization operators is at most 52%. For example, at the first iteration Gen_P can prove less properties than Gen_{MH}, but it outperforms the monovariant operator with convex hull by recovering precision when iterating the specialization.

The increase of precision due to the iteration of program specialization is also higher for generalization operators that do not use the convex hull operator (Gen_M and Gen_P), compared to their counterparts that use convex hull (Gen_{MH} and Gen_{PH}).

We would also like to point out that the use of convex hull is very useful during the first iterations. Indeed, the generalization operators using convex hull can verify 104 programs at the first iteration, while operators not using convex hull can verify 74 programs only. In this case the choice of using a polyvariant vs. monovariant generalization operator has no effect on the number of verified programs at the first iteration.

Finally, we note that the sets of programs with correct answers by using different operators are not always comparable. Indeed, the total number of different programs with correct answers by using any of the generalization operators we have considered is 190, while the programs for which a single operator produced a correct answer is at most 185. Moreover, there are programs that can be verified by operators with lower precision but that cannot be verified by operators with higher precision, within the considered timeout limit. For example, in our experiments the least precise operator Gen_M was able to prove four programs for which the most precise operator Gen_{PH} timed out.

This confirms that different generalization operators can give, in general, different results in terms of precision and performance. If we do not consider time limitations, generalization operators having higher precision should be preferred over less precise ones, because they may offer more opportunities to discover the invariants that are useful for proving the properties of interest. In some cases, however, the use of more precise generalization operators determines the introduction of more definition clauses, each requiring an additional iteration of the while-loop (α) of the *Specialize* Procedure, thereby slowing down the verification process and possibly preventing the termination within the considered timeout limit. In practice, the choice of the generalization operator to be used for any given verification problem at hand, can be made according to some heuristics that may be provided on the basis of the above mentioned trade-off between precision and efficiency.

8. Related Work and Conclusions

The software model checking technique proposed in this paper is an extension of the technique for the verification of simple imperative programs presented in [12]. The main improvements and novelties introduced in this work are the following: (i) we have considered a significant fragment of the C Intermediate Language [38], (ii) we have defined a general verification framework in which specialization of constraint logic programs is iteratively applied with the objective of using in a more effective way the information dispersed throughout the program to be verified (and, in particular, the information associated with the constraints in the initial configurations and the error configurations), and (iii) we have performed an extensive experimental evaluation on a large set of examples (over 200) taken from different sources, and we have shown that our approach, despite its generality, is effective and efficient in practice. The use of iterated specialization avoids the least model construction performed after program specialization by the technique presented in [12].

The fragment of the programming language we have considered has several limitations. In particular, we have assumed that: (i) all variables have integer type, (ii) neither arrays nor structured data nor pointers are present, and (iii) there are no definitions of recursive functions. However, the approach we have proposed here is very general and we believe that it can be followed also in the case of much richer programming languages. Indeed, an application of our approach to programs with arrays appears in [14].

The use of constraint-based techniques for program verification has recently received a renewed attention [4, 43], and in particular, constraints turn out to be very suitable for expressing both the symbolic executions of imperative programs and their execution invariants. As an evidence of the expressive power of constraints we want to point out that *Constrained Horn Clauses* (which basically are Constraint Logic Programs) have been recently proposed in [4] as a common intermediate language for exchanging program properties between software verifiers. In the same line of work, [24] presents a method for the automatic synthesis of software verification tools that use rules for reachability and termination proofs written in a formalism similar to Horn clauses.

Our work is related to that of [4, 43], because we also use Horn clauses and constraints, and we also propose a general approach to program verification. In particular, the technique we have presented in this paper can be seen as an application of a general verification method based on CLP and program transformation. In that general method CLP is used for specifying: (i) the programming language under consideration and its semantics, and (ii) the logic used for expressing the properties of interest and its proof rules, and CLP program transformation is used as a general-purpose engine for analysis. By modifying the clauses that specify the interpreter of the language in which the programs to be verified are written, one can encode, in an agile way, the semantics of different languages either logic, or functional, or concurrent ones. Also the class of the properties to be verified, which in this paper is restricted to reachability properties, can be extended to those specified by more expressive logics, such as the Computational Tree Logic used in [21] for the verification of infinite state reactive systems.

The use of program specialization for the verification of properties of imperative programs is not novel. It has been considered, for instance, in [40]. In that paper a CLP interpreter for the operational semantics of a simple imperative language is specialized with respect to the input program to be verified. Then, a static analyzer for CLP programs is applied to the residual program for computing invariants (that is, overapproximations of the behavior) of the input imperative program. These invariants are used in the proof of the properties of interest. Unlike [40], our verification method does not perform any static analysis phase separated from the specialization phase and, instead, we discover program invariants during the specialization process by applying suitable generalization operators. These operators are defined in terms of operators and relations on constraints such as widening and convex hull [7, 10, 21]. As in [40], we also use program specialization to perform the so-called removal of the interpreter, but in addition, in this paper we repeatedly use specialization for propagating the information about the constraints that occur in the initial configurations and in the error configurations.

The specialization of logic programs and constraint logic programs has also been considered in various techniques for the verification of infinite state reactive systems [19, 21, 35]. By using these techniques one may verify properties of Kripke structures, instead of properties of imperative programs as we did in this paper. In [35] the authors do not make use of constraints, which are a key ingredient of the technique we present here. In [19, 21, 35] program specialization is followed by the computation of the least model of the specialized program, or the computation of an overapproximation of the least model via abstract interpretation. In this paper we have replaced the phase in which the least model (or an over approximation of it) is computed, by a phase in which program specialization is iterated.

A popular technique for program verification is the Counter-Example Guided Abstraction Refinement (CEGAR) [6, 32, 44], which is used by many software model checkers such as BLAST [3], DAGGER [25], and SLAM [1]. In the CEGAR technique, given a program P and a safety property to be verified, one automatically constructs an abstract model of P which is used to check whether or not an abstract error configuration is reachable from an abstract initial configuration. If no abstract error configuration can be reached, then P satisfies the given safety property, otherwise a counterexample, that is, a sequence of configurations leading to an abstract error configuration, is generated and then analyzed. If the counterexample corresponds to a concrete computation of P, then the program is proved unsafe, otherwise the abstraction needs to be refined because it was too coarse, and a new cycle of the verification process is started by using that refined abstraction to get a new abstraction of the program P, in the hope of a successful proof.

Our approach can be regarded as complementary to the approaches based on CEGAR. Indeed, we begin by making no abstraction at all, and if the specialization process is deemed to diverge, then we perform some generalization steps which play a similar role to that of abstraction. (Note, however, that unlike abstraction program specialization preserves program equivalence.) There are various generalization operators that we can apply for that purpose and by varying those operators we can tune the specialization process in the hope of making it more effective for the proofs of the properties of interest. Moreover, since our specialization-based method preserves the semantics of the original specification, we can apply a *sequence* of specializations, thereby refining the analysis in successive steps and, hopefully, improving the level of precision.

In the field of static program analysis the idea of performing backward and forward semantic analyses has been proposed in [8]. These analyses have been combined (see, for instance, [9]) to devise a fixpoint-guided abstraction refinement algorithm which has been proved to be at least as powerful as the CEGAR algorithm where the refinement is performed by applying a backward analysis. An enhanced version of the algorithm presented in [9], which improves the abstract state space exploration and makes use of disjunctive abstract domains, has been proposed in [42]. In the present paper we have shown that also in our approach the idea of iterating program analysis and traversing the graph of the configurations both in the forward and backward manner can be fruitfully exploited, once we have reduced the program analysis task to a program transformation task.

As future work, we would like to develop, within the general framework based on CLP transformation, verification techniques that support other language features, including recursive function calls, concurrency, and more complex data types, such as data structures and pointers. This will also allow us to enlarge the set of examples used in the experimental evaluation by considering real-world programs. At the same time, we would like to extend our approach to the proof of more complex properties that, for instance, depend on the content of the data structures (such as the values of the elements of arrays or lists). As already mentioned, a step forward in this direction has been done in the case of programs manipulating arrays [14]. The most relevant ingredient needed to handle array programs is a new transformation rule that allows us to manipulate arrays via predicates which represent array operations and satisfy suitable axioms [36]. We think that by a similar approach one may be able to deal also with different, more complex data structures.

In the future we also wish to develop, within our specialization approach, some techniques that guarantee a form of progression of the analysis similar to the one achieved by the CEGAR approach, whereby a spurious counterexample, once discarded by a suitable abstraction, does not occur again in successive refinements.

Another interesting direction to explore is the application of our method to termination proofs of imperative programs.

Finally, since program specialization produces an equivalent program (with respect to the validity of the property of interest), we can also investigate the use of program specialization as a preprocessing step before using other software model checkers with the aim of improving their precision and efficiency.

9. Acknowledgments

We thank the anonymous referees of PEPM 2013 and of Science of Computer Programming for their very helpful suggestions and constructive comments.

References

[1] T. Ball, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[2] D. Beyer. Second competition on software verification (Summary of SV-COMP 2013). In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '13*, Lecture Notes in Computer Science 7795, pages 594–609. Springer, 2013.

[3] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, 2007.

[4] N. Bjørner, K. McMillan, and A. Rybalchenko. Program verification as satisfiability modulo theories. In *Proceedings of the 10th International Workshop on Satisfiability Modulo Theories, SMT-COMP '12*, pages 3–11, 2012.

[5] D. R. Brough and C. J. Hogger. Grammar-related transformations of logic programs. *New Generation Computing*, 9(1):115–134, 1991.

[6] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification, CAV '00*, Lecture Notes in Computer Science 1855, pages 154–169. Springer, 2000.

[7] 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, 1977.

[8] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL '79)*, pages 269–282. ACM Press, 1979.

[9] P. Cousot, R. Ganty, and J.-F. Raskin. Fixpoint-Guided Abstraction Refinements. In *Proceedings of the 14th International Symposium on Static Analysis, SAS '07*, Lecture Notes in Computer Science 4634, pages 333–348. Springer, 2007.

[10] 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, 1978.

[11] B. Cui and D. S. Warren. A system for tabled constraint logic programming. In J. W. Lloyd, editor, *Proceedings of the First International Conference on Computational Logic, CL '00, London, UK, 24-28 July*, Lecture Notes in Artificial Intelligence 1861, pages 478–492. Springer-Verlag, 2000.

[12] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Specialization with constrained generalization for software model checking. In *Proceedings of the 22nd International Symposium Logic-Based Program Synthesis and Transformation, LOPSTR '12*, volume 7844 of *Lecture Notes in Computer Science*, pages 51–70. Springer, 2013.

[13] 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, New York, NY, USA, 2013. ACM.

[14] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying Array Programs by Transforming Verification Conditions. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '14*, volume 8318 of *Lecture Notes in Computer Science*, pages 182–202. Springer, 2014.

[15] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. VeriMAP: A Tool for Verifying Programs through Transformations. In *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS '14*, volume 8413 of *Lecture Notes in Computer Science*, pages 568–574. Springer, 2014.

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

[17] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In K.-K. Lau, editor, *Proceedings of the Tenth International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '00, London, UK, 24-28 July, 2000*, Lecture Notes in Computer Science 2042, pages 125–146. Springer-Verlag, 2001.

[18] F. Fioravanti, A. Pettorossi, and M. Proietti. Transformation rules for locally stratified constraint logic programs. In K.-K. Lau and M. Bruynooghe, editors, *Program Development in Computational Logic*, Lecture Notes in Computer Science 3049, pages 292–340. Springer-Verlag, 2004.

[19] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving reachability analysis of infinite state systems by specialization. In G. Delzanno and 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, pages 165–179. Springer, 2011.

[20] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Using real relaxations during program specialization. In G. Vidal, editor, *Logic-Based Program Synthesis and Transformation - 21st International Symposium, LOPSTR '11, Odense, Denmark, July 18-20, 2011. Revised Selected Papers*, Lecture Notes in Computer Science 7225, pages 106–122. Springer, 2012.

[21] 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.

[22] J. P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of the 1993 ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation, PEPM '93, Copenhagen, Denmark*, pages 88–98. ACM Press, 1993.

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

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

[25] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. 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, pages 443–458. Springer, 2008.

[26] A. Gupta and A. Rybalchenko. InvGen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification, CAV '09*, Lecture Notes in Computer Science 5643, pages 634–640. Springer, 2009.

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

[28] J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.

[29] J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.

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

[31] J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In I. Gent, editor, *Principles and Practice of Constraint Programming, CP '09*, Lecture Notes in Computer Science 5732, pages 454–469. Springer, 2009.

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

[33] 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.

[34] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.

[35] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialization. In A. Bossi, editor, *Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation, LOPSTR '99, Venezia, Italy*, Lecture Notes in Computer Science 1817, pages 63–82. Springer, 2000.

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

[37] S. P. Miller, M. W. Whalen, and D. D. Cofer. Software model checking takes off. *Commun. ACM*, 53(2):58–64, 2010.

[38] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In R. Horspool, editor, *Compiler Construction*, Lecture Notes in Computer Science 2304, pages 209–265. Springer, 2002.

[39] J. C. Peralta and J. P. Gallagher. 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, pages 90–108. Springer, 2003.

[40] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, editor, *Proceedings of the 5th International Symposium on Static Analysis, SAS '98*, Lecture Notes in Computer Science 1503, pages 246–261. Springer, 1998.

[41] A. Podelski and A. Rybalchenko. 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, pages 245–259. Springer, 2007.

[42] F. Ranzato, O. Rossi-Doria, and F. Tapparo. A forward-backward abstraction refinement algorithm. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI '08*, Lecture Notes in Computer Science 4905, pages 248–262. Springer, 2008.

[43] A. Rybalchenko. Constraint solving for program verification: Theory and practice by example. In T. Touili, B. Cook, and P. Jackson, editors, *Proceedings of the 22nd International Conference on Computer Aided Verification, CAV '10*, Lecture Notes in Computer Science 6174, pages 57–71. Springer, 2010.

[44] H. Saïdi. Model checking guided abstraction and analysis. In *Proceedings of the 7th International Symposium on Static Analysis, SAS '00*, Lecture Notes in Computer Science 1824, pages 377–396. Springer, 2000.