# Verifying Programs via Iterated Specialization

Emanuele De Angelis,
Fabio Fioravanti
University G. D'Annunzio,
Pescara, Italy
{deangelis,fioravanti}@sci.unich.it

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

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

## 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 C programs with integer variables and we focus our attention on safety properties, stating that no error configuration can be reached from the initial configurations. We encode the interpreter of the language as a CLP program $I$, and we also encode the safety property to be verified as the negation of a predicate *unsafe* defined in $I$. Then, we specialize the CLP program $I$ with respect to the given C program and the given initial and error configurations, with the objective of deriving a new CLP program $I_{sp}$ which either contains the fact *unsafe* (and in this case the C program is proved unsafe) or contains no clauses with head *unsafe* (and in this case the C program is proved safe). If $I_{sp}$ does not enjoy this property 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, or propagating backward from an error configuration) and different operators (such as widening and convex hull operators) for generalizing predicate definitions. Due to the undecidability of program safety, the iterated specialization process may not terminate. By an experimental evaluation carried out on a set of examples taken from the literature, we show that our method is competitive with respect to state-of-the-art software model checkers.

***Categories and Subject Descriptors*** I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation, Program verification; F.3.1 [*Logic and Meaning of Programs*]: Semantics of Programming Languages—Partial evaluation, Program analysis; F.3.2 [*Logic and Meaning of Programs*]: Specifying and Verifying and Reasoning about Programs—Mechanical verification; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Logic and constraint programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods, Model checking

***General Terms*** Languages, Theory, Verification

***Keywords*** Software model checking, constraint logic programming, program specialization

## 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 [25] 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 (a recent survey is presented in [20]).

In this paper we consider programs acting on integer variables which belong to a subset of CIL, the C Intermediate Language [26], called *Core CIL*. Then we address the problem of verifying *safety* properties, stating that when executing a program, an unsafe configuration cannot be reached from any initial configuration.

Since safety is an undecidable problem for programs that act on integer numbers, many program analysis techniques follow approaches based on *abstraction* [5], by which the integer data domain is mapped to an abstract domain so that reachability is preserved, that is, 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 unsafe configuration is not reachable from an abstract initial configuration, then the program is proved to be safe (see [20] for a general abstract reachability algorithm). Notable abstractions are those based on convex polyhedra, that is, conjunctions of linear inequalities (also called *constraints* here).

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, [17, 19, 28, 29]). In the context of CLP-based program analysis, program specialization has been proposed as a means for translating an imperative program to CLP [28]. By following the approach presented in [28], the semantics of an imperative language is defined by means of a CLP program which is the interpreter $I$ of that language. Then, the interpreter $I$ is specialized with respect to the input program $P$ whose safety property should be checked. The result of this specialization is a CLP program $I_{sp}$ and, since program specialization preserves semantic equivalence, for proving properties of the imperative program $P$, we can analyze the CLP program $I_{sp}$ by applying the above mentioned techniques based on polyhedral abstractions.

It has also been pointed out that program specialization can be used as a technique for software model checking on its own [9]. 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 program $I'_{sp}$ whose least model $M(I'_{sp})$ can be computed in finite time because $I'_{sp}$ can be represented by a finite 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 which always terminates and produces a specialized program whose least model can be finitely computed.

In order to mitigate this limitation, in this paper we propose a method based on the repeated application of program specializations, called here *iterated specialization*.

By repeated program specializations we can produce a sequence of 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 program and explicitly adding new constraints that denote invariants of the computation. Thus, the effect of iterated specialization is the propagation of these constraints from one program version to the next, and iterated specialization terminates when a program with finite least model is generated. In general, we have no guarantee that iterated specialization terminates.

The paper is organized as follows.

In Section 2 we describe the syntax of the Core CIL language and the CLP interpreter which defines its operational semantics. In Section 3 we specify the problem of proving program safety we want to address. In Section 4 we outline our software model checking approach to establish program safety by presenting a simple example. In Section 5 we describe the overall strategy of iterated specialization, and also some specific strategies for performing single specialization steps. In Section 6 we report on some experiments we have performed by using a prototype implementation based on the MAP transformation system [24]. We also compare the results we have obtained using the MAP system with the results we have obtained using state-of-the-art software model checking systems such as ARMC [29], HSF(C) [13], and TRACER [18]. Finally, in Section 7 we discuss the related work and, in particular, we compare our approach with other existing methods of software model checking.

## 2. A CLP Interpreter for the Language CIL

We assume that the programs to be verified are written in a subset of CIL, called Core CIL. Here is its syntax, where: (i) *Vars* is a set of variable identifiers, (ii) *Ids* is a set of identifiers for types, and (iii) $\mathbb{Z}$ is the set of integers which denote the constants of the basic types (such as the integer constants, the character constants, etc.)

| | | |
|---|---|---|
| $x, y, \ldots$ | $\in$ | *Vars* (variable identifiers) |
| $f, g, \ldots$ | $\in$ | Function identifiers |
| $\ell, \ell_1, \ldots$ | $\in$ | Labels |
| *id* | $\in$ | *Ids* (identifiers of types) |
| *const* | $\in$ | $\mathbb{Z}$ (integer constants, character constants, ...) |
| *basic* | $\in$ | Basic types (`int`, `char`, ...) |
| *uop*, *bop* | $\in$ | Unary and binary operators (`+`, `-`, `<=`, ...) |

$$\begin{array}{lll}
\text{prog} & ::= & \text{typedef}^* \ \text{decl}^* \ \text{fundef}^* \\
\text{typedef} & ::= & \texttt{typedef}\ \text{type}\ id \\
\text{decl} & ::= & \text{type}\ id \mid \text{type}\ id\ [const] \\
\text{type} & ::= & basic \mid id \\
\text{fundef} & ::= & \text{type}\ f\ (\text{decl}^*)\ \{\text{decl}^*\ \ \text{lab\_cmd}^+\} \\
\text{lab\_cmd} & ::= & \ell : \text{cmd} \\
\text{cmd} & ::= & \texttt{halt} \mid x{=}\text{exp} \mid x{=}f(\text{exp}^*) \\
& & \mid\ \texttt{if}\ (\text{exp})\ \ell_1\ \texttt{else}\ \ell_2 \\
& & \mid\ \texttt{goto}\ \ell \mid \texttt{return}\ \text{exp} \\
\text{exp} & ::= & const \mid x \mid uop\ \text{exp} \\
& & \mid\ \text{exp}\ bop\ \text{exp} \mid (\text{type})\ \text{exp}
\end{array}$$

For reasons of brevity, we will feel free to say ‘command’, instead of ‘labelled command’. The elements of a sequence denoted by an expression of the form $e^*$ or $e^+$ will be separated by semicolons. Note that `while` commands can be replaced, as usual, by suitable `if-else` and `goto` commands. We assume that every label occurs in every program at most once. The sequence $\text{fundef}^*$ of function definitions is usually not empty and it contains the definition of the function `main`.

Now we give the semantics of Core CIL. For that purpose let us first introduce the following auxiliary functions and data structures. We also assume that: (i) every variable occurrence is either global or local to a function definition, (ii) in every given program for every variable occurrence $x$, one may statically determine whether $x$ is a global variable or a local variable, and (iii) there is only one level of locality (that is, there are no blocks and thus, no nested levels of locality).

A *global environment* $\delta\colon Vars \rightarrow \mathbb{Z}$ is a function which maps global variables to their integer values. Likewise, a *local environment* $\sigma\colon Vars \rightarrow \mathbb{Z}$ maps function parameters and local variables to their integer values.

An *activation frame* is a 3-tuple $\langle \ell, y, \sigma \rangle$, where: (i) $\ell$ is the label where to jump after returning from a function call, (ii) $y$ is the variable where to store the value returned by a function call, and (iii) $\sigma$ is the local environment to be initialized when making a call binding the function parameters and local variables.

A *configuration* is a 3-tuple of the form $\langle\!\langle c, \delta, \tau \rangle\!\rangle$ where: (i) $c$ is a labelled command, (ii) $\delta$ is a global environment, and (iii) $\tau$ is a list of activation frames. We operate on the list $\tau$ by the usual head ($hd$) and tail ($tl$) functions and the right-associative constructor cons ($:$). The empty list is denoted by $[\,]$. By $update(f, x', v')$ we denote the function $f'$ such that if $x = x'$ then $f'(x) = v'$ else $f'(x) = f(x)$. For any program $P$, for any label $\ell$, (i) $at(\ell)$ denotes the command in $P$ with label $\ell$, and (ii) *nextlab*($\ell$) denotes the label of the command in $P$ which is written *immediately after* the command with label $\ell$. Given a function identifier $f$, $at(f)$ denotes the first labelled command of the definition of the function $f$. For any expression $e$, global environment $\delta$, and local environment $\sigma$, $[\![e]\!]\delta\sigma$ is the integer value of $e$. We assume that the evaluation of expressions has no side effects.

The operational semantics (that is, the interpreter) of the Core CIL language is given as a transition relation $\Longrightarrow$ between configurations according to the following rules $R1$–$R5$. Notice that no rules are given for the command $\ell$ : `halt`. Thus, no new configuration is constructed when the command of the configuration at hand is `halt`.

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

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

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

Informally, an assignment updates either the global environment or the local environment of the topmost activation frame.

($R2$). *Function call.* Let $hd(\tau)$ be $\langle \ell', y, \sigma \rangle$. 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 $f$.

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

where $\overline{\sigma}$ is a set of bindings of the form:

$\overline{\sigma} = \{\langle x_1, [\![e_1]\!]\delta\sigma \rangle, \ldots, \langle x_k, [\![e_k]\!]\delta\sigma \rangle, \langle y_1, n_1 \rangle, \ldots, \langle y_h, n_h \rangle\}$

for some values $n_1, \ldots, n_h$ in $\mathbb{Z}$ (indeed, when we declare the local variables we do not initialize them). 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 the label where to jump after returning from the call, the variable where to store the returned value, and the new local environment.

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

If $y$ is a global variable:

$\langle\!\langle \ell : \texttt{return}\, e, \delta, \tau \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell'), update(\delta, y, v), tl(\tau) \rangle\!\rangle$

If $y$ is a local variable:

$\langle\!\langle \ell\!:\! \texttt{return}\, e, \delta, \tau \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell'), \delta, \langle \ell'', z, update(\sigma', y, v)\rangle : \tau'' \rangle\!\rangle$

Informally, a return command erases the topmost activation frame and updates either the global or the local environment of the new topmost activation frame.

$(R4)$. *Goto.* $\langle\!\langle \ell : goto\ \ell', \delta, \tau \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell'), \delta, \tau \rangle\!\rangle$

$(R5)$. *If-then-else.* Let $hd(\tau)$ be $\langle \ell', y, \sigma \rangle$.

If $[\![e]\!]\delta\sigma = true$:

$\langle\!\langle \ell : \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2, \delta, \tau \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell_1), \delta, \tau \rangle\!\rangle$

If $[\![e]\!]\delta\sigma = false$:

$\langle\!\langle \ell : \texttt{if}\ (e)\ \ell_1\ \texttt{else}\ \ell_2, \delta, \tau \rangle\!\rangle \Longrightarrow \langle\!\langle at(\ell_2), \delta, \tau \rangle\!\rangle$

The *initial configuration* is the 3-tuple (without loss of generality, we assume that the function `main` has no arguments):

$\langle\!\langle \ell_{init} : z_0 = \texttt{main}(),\ \delta_{init},\ [\,] \rangle\!\rangle$

where:

(i) $\ell_{init}$ is a fresh new label such that $nextlab(\ell_{init})$ is a fresh new label $\ell_{halt}$ whose associated command is `halt` (indeed, no command should be executed after the function call: $z_0 = \texttt{main}()$),

(ii) $z_0$ is a fresh new global variable (whose type should comply with those of the expressions in the `return` commands of the function `main`()), and

(iii) $\delta_{init}$ is the initial global environment which is of the form: $\{\langle z_1, n_1\rangle, \ldots, \langle z_r, n_r\rangle\}$, where $z_1, \ldots, z_r$ are the global variables of the given program and $n_1, \ldots, n_r$ are some given values in $\mathbb{Z}$.

Note that initially the list of activation frames is empty and the first activation frame is constructed when executing the initial function call: $z_0 = \texttt{main}()$. That frame is of the form: $\langle \ell_{halt}, z_0, \overline{\sigma}\rangle$, where $\overline{\sigma}$ is a local environment binding the local variables of the definition of the function `main`().

The semantics we have given above can be extended to a larger subset of CIL which includes in particular array and structure types.

Let us now recall some notions and terminology concerning constraint logic programming. For more details the reader may refer to [16]. If $\texttt{p}_1$ and $\texttt{p}_2$ are linear polynomials whose variables and coefficients are of type `int`, then $\texttt{p}_1 = \texttt{p}_2$, $\texttt{p}_1 \geq \texttt{p}_2$, and $\texttt{p}_1 > \texttt{p}_2$ are *atomic constraints*. A *constraint* is either *true*, or *false*, or an atomic constraint, or a *conjunction* of constraints. A CLP program is a finite set of clauses of the form `A:- c,B`, where `A` is an atom, `c` is a constraint, and `B` is a (possibly empty) conjunction of atoms. The clause `A:- c` is called a *constrained fact*.

The semantics of a CLP program $P$ is defined to be the *least model* of $P$, denoted $M(P)$, which agrees with the standard interpretation on the integers for the constraints.

The CLP interpreter for our Core CIL language is given by the following clauses for the binary predicate `tr` which relates old configurations to new configurations and defines the transition relation $\Longrightarrow$.

```
1. tr(cf(cmd(L,asgn(X,E)),D,T),cf(cmd(L1,C),D1,T)) :-
       loc_env(T,S), aeval(E,D,S,V), update(D,X,V,D1),
       nextlab(L,L1), at(L1,C).
2. 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).
3. 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).
4. tr(cf(cmd(L,goto(L1)),D,T),cf(cmd(L1,C),D,T)) :-
       at(L1,C).
```

The term `asgn(X,E)` encodes the assignment to a global variable of the form $x = e$. Similarly, the terms `ite(E,L1,L2)` and `goto(L)` encode the conditional `if` $(e)\,\ell_1$ `else` $\ell_2$ and the jump `goto` $\ell$, respectively. The term `cmd(L,C)` encodes the command `C` with label `L`. The predicate `loc_env` extracts the local environment from the topmost activation frame in a configuration. The predicate `aeval(E,D,S,V)` computes the value `V` of the arithmetic expression `E` in the global environment `D` and the local environment `S`. Likewise 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 `update(D,X,V,D1)` updates the global environment `D`, thereby constructing the new global environment `D1`, by binding the variable `X` to the value `V`. The predicate `at(L,C)` binds to `C` the command with label `L`. The predicate `nextlab(L,L1)` binds to `L1` the label of the command which is written immediately after the command with label `L`.

We have listed the clauses for the cases of: (i) assignment to global variables (clause 1), (ii) `if-else` (clauses 2 and 3), and (iii) `goto` commands (clause 4), because they are the only cases of interest in our examples below. The definition of `tr` for the cases of assignment to local variables, function call and `return` are similar.

Notice that the CLP clauses 1–4 for the predicate `tr` have no constraints in their bodies. However, constraints are used in the definitions of the predicates `aeval` and `beval`.

## 3. The safety problem

In this paper we consider the problem of verifying the *safety* of program fragments. Then, safety of programs will be defined in terms of safety of program fragments. A *program fragment* is a (possibly empty) program followed by a non-empty sequence of labelled commands. Thus,

prog_fragm ::= prog lab_cmd$^+$ (†)

The problem of verifying the safety of a program fragment $P$ 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 fragment with 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.

Without loss of generality, we assume that the last command of $P$ is $\ell_h$ :`halt` and no other `halt` command occurs in $P$.

We say that a program fragment $P$ 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 $\varphi_{init}$ and $\varphi_{error}$, if there exist global environments $\delta_{init}$ and $\delta_h$ such that:

(i) $\varphi_{init}(\delta_{init}(z_1), \ldots, \delta_{init}(z_r))$ holds and

(ii) $\langle\!\langle \ell_0 : c_0, \delta_{init}, [\,] \rangle\!\rangle \Longrightarrow^* \langle\!\langle \ell_h : \texttt{halt}, \delta_h, [\,] \rangle\!\rangle$ and

(iii) $\varphi_{error}(\delta_h(z_1), \ldots, \delta_h(z_r))$ holds,

where $\ell_0 : c_0$ is the first command in the sequence lab_cmd$^+$ of labelled commands at the right end of $P$ (see (†) above).

A program fragment is said to be *safe* with respect to $\varphi_{init}$ and $\varphi_{error}$ iff it is not unsafe with respect to $\varphi_{init}$ and $\varphi_{error}$.

We define the unsafety (and safety) of a program $P'$ with respect to the formulas $\varphi_{init}$ and $\varphi_{error}$ as the unsafety (and safety, respectively) of the program fragment obtained from $P'$ by: (i) deleting the function `main`(), and (ii) adding at the right end of $P'$ the sequence of labelled commands of the function `main`(), where the command `return` $e$ has been replaced by $\ell_h$ : `halt` (note that, without loss of generality, we may assume that in $P'$ there is a single `return` command).

When ambiguity does not arise, we will feel free to say 'program', instead of 'program fragment'.

An unsafety triple can be encoded as a CLP program. We show how to do this encoding through the following example. The extension to the general case is straightforward and will be omitted. Let us consider the unsafety triple:

$\{\!\{\varphi_{init}(x,y,n)\}\!\}\ P\ \{\!\{\varphi_{error}(x,y,n)\}\!\}$ where

$\varphi_{init}(x,y,n)$ is $x=0 \wedge y=0$
$P$ is $\ell_0$: `while` $(x<n)\ \{x=x+1; y=x+y;\}$;
$\ell_h$: `halt`
$\varphi_{error}(x,y,n)$ is $x>y$

(In this program fragment $P$ we have an empty program followed by the above two commands `while` and `halt`.)

First, we replace the `while` command by the following sequence of Core CIL commands:

$\ell_0$: `if` $(x<n)\ \ell_1$ `else` $\ell_h$
$\ell_1$: $x=x+1$
$\ell_2$: $y=x+y$
$\ell_3$: `goto` $\ell_0$
$\ell_h$: `halt`

Then, this sequence of commands is translated into the following CLP facts:

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

We also have the following clauses that specify the reachability relation from the initial configuration to the error configuration:

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

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

```
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.
```

In the initial configuration (see clause 9) we have the initial command `cmd(0, ite(less(int(x), int(n)), 1, h))` and the initial list of activation frames which is empty. In clauses 9 and 10 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]]` which gives the bindings of the variables $x$, $y$, and $n$, respectively.

The CLP program consisting of clauses 1–10 above, together with the clauses that define the predicate `tr` (see clauses 1–4 of Section 2), is called the *CLP encoding* of the given unsafety triple $\{\!\{\varphi_{init}(x,y,n)\}\!\}\ P\ \{\!\{\varphi_{error}(x,y,n)\}\!\}$.

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

## 4. The Software Model Checking Method in Action

In this section we present an application of our software model checking method based on *iterated specialization*, which performs a *sequence* of program specializations, rather than one specialization only. The formal presentation of the method will be given in the next section.

In the example we will consider the *iteration* of program specializations plays a crucial role and is required for the proof of program safety.

Let us consider the unsafety triple of the previous section. We want to show that the program fragment $P$ is safe with respect to $\varphi_{init}(x,y,n)$ and $\varphi_{error}(x,y,n)$.

Our method for proving program safety consists of three specialization steps: (i) the removal of the interpreter (this step is common to other specialization-based techniques for the verification of imperative programs [9, 28]), (ii) the propagation of the constraints of the initial configuration, and (iii) the propagation of the constraints of the error configuration.

### First Specialization: Removal of the interpreter

We start off from the CLP clauses 1–10 associated with the given program fragment $P$ (see Section 3), and the CLP clauses for the predicate `tr` (clauses 1–4 of Section 2) which define the interpreter of the Core CIL language. At the end of this first specialization we will derive a CLP program (see program $P1$ below) which evaluates the predicate `unsafe` without evaluating the predicate `tr`. In this sense we say that this first specialization realizes the removal of the interpreter.

In order to get such a CLP program we specialize clauses 6–8 with respect to the given definitions of the predicates `initConf`, `errorConf`, `tr`, and `at`. This specialization is performed by following the usual *unfold-definition-fold cycle* of the rule-based specialization strategies [11]. In particular, we will follow the Specialization procedure presented in Figure 2 of Section 5.

The various specialization steps are performed in an automatic way by our MAP system [24].

We start off by unfolding clause 6 with respect to the atom `initConf(X)` and 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]], [ ])).
```

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]], [ ])).
```

We fold clause 11 and we get:

```
11.f unsafe :- X = 0, Y = 0, new1(X, Y, N).
```

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]], [ ])).
```

From clause 13, after a few unfolding steps which perform the symbolic evaluation of the `if-then` command using the clause for the predicate `tr`, we get the following two clauses:

```
15. new1(X, Y, N) :- X < N,
        reach(cf(cmd(1, asgn(int(x), 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 in clause 13 generates 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, ...)`).
From clause 15, after two unfolding steps, we get:

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

From clause 16, 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]],[ ])).
```

At this point the program at hand is made out of clauses 11.f, 17, and 18. Then, by unfolding clause 17, we get:

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

By unfolding clause 18 we get:

```
20. new1(X,Y,N) :- X≥N, X>Y.
```

From clause 19, after two unfolding steps, we get:

```
21. new1(X,Y,N) :- X<N, X1=X+1,
        tr(cf(cmd(1,asgn(int(y),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 12, 15, 19, and 22, which we have obtained by unfolding, mimics the execution of the sequence of the four commands: (i) $\ell_0$ : `if` $(x<n)\ \ell_1$ `else` $\ell_h$, (ii) $\ell_1 : x = x+1$, (iii) $\ell_2 : y = x+y$, and (iv) $\ell_3$ : `goto` $\ell_0$ (note in those clauses the atoms `reach(cf(cmd(`$i$`,...),...,...))`, for $i = 0,1,2,3$). Indeed, in general, by unfolding, one is able to perform the symbolic execution of the commands of any given program. The conditions that should hold so that a particular command `cmd(`$i$`,...)` is executed, are given by the constraints in the clause in whose body the atom `reach(cf(cmd(`$i$`,...),...,...))` occurs.

From clause 22, after a few 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]],[ ])).
```

By folding clause 23 using clause 12, we get:

```
23.f new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
```

Note that this folding step using the definition for the predicate `new1` is possible because the execution of the program returned to the command to which the definition of `new1` refers. The final, specialized program $P1$ is as follows:

```
11.f unsafe :- X=0, Y=0, new1(X,Y,N).
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.
```

The derived program $P1$ has a constrained fact for `new1` (see clause 20 and, by repeatedly using clause 23.f, from that constrained fact, we can derive infinitely many new constrained facts which belong to the least model of $P1$. Hence, we cannot show that `new1` does not hold for `X=Y=0`, and thus we cannot show that `unsafe` does not hold (see clause 11.f).

In order to show program safety, now we perform two more specialization steps. First, we specialize program $P1$ by with respect to the constraints of the initial configuration, and then we specialize the residual program with respect to the constraints of the error configuration. By iterated specialization we will derive a new *empty* program allowing us to conclude that `unsafe` does not hold.

**Second Specialization: Propagation of the constraints of the initial configuration**

Now we perform our second program specialization starting from the program $P1$ we have derived by removing the interpreter. This specialization is based on the idea of propagating the constraints `X=0` and `Y=0` of the initial configuration which occur in clause 11.f defining the predicate `unsafe`.

We begin by unfolding clause 11.f with respect to the atom with predicate `new1` and we get:

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

Now clause 24 has an unsatisfiable constraint, and thus it is deleted. In order to fold clause 25, we define the following new predicate:

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

By folding clause 25, we get:

```
25.f. unsafe :- N>0, X1=1, Y1=1, new2(X1,Y1,N).
```

Now we unfold the last definition which has been introduced (clause 26) and we get two clauses of which the only one with a satisfiable constraint is (after constraint simplification):

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

In order to fold this clause, we need the following new predicate:

```
28. new3(X,Y,N) :- N>1, X=2, Y=3, new1(X,Y,N).
```

The comparison between clauses 26 and 28 shows the risk of introducing an infinite number of clauses (see, in particular, the constraints `X=1` and `X=2`), thereby making the specialization process never to halt. Thus, we perform a generalization step (we use the *widening* operator [8]) between clauses 26 and 28, and we introduce, instead of clause 28, the following clause 29 (where the constraint `X≥1` is the widening of `X=1` and `X=2`):

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

We fold clause 27 using clause 29 and we get (after constraint simplification):

```
27.f new2(X,Y,N) :- X=1, Y=1, N>1, X1=2, Y1=3,
                    new3(X1,Y1,N).
```

Note that this folding step preserves equivalence between clauses 27 and 27.f, even if `new3` is a generalization of `new1`, because the atoms `new1(X1,Y1,N)` and `new3(X1,Y1,N)` are equivalent in a context where the constraint `N>1, X1=2, Y1=3` holds.

By continuing our specialization process following the usual unfold-definition-fold cycle according to the Specialization procedure of Figure 2, we eventually get the specialized program $P2$:

```
25.f unsafe :- N>0, X1=1, Y1=1, new2(X1,Y1,N).
27.f new2(X,Y,N) :- X=1, Y=1, N>1, X1=2, Y1=3,
                    new3(X1,Y1,N).
30.  new3(X,Y,N) :- X1≥1, Y1≥X1, X1≤N,
                X1=X+1, Y1=X1+Y, new3(X1,Y1,N).
31.  new3(X,Y,N) :- Y≥1, N>0, X≥N, X>Y.
```

Again, as after the removal of the interpreter, in this final program the presence of a constrained fact for the predicate `new3` (see clause 31), does not allow us to conclude that $P2$ has an empty least model, and hence the safety of our program.

**Program Reversal**

At this point the novel strategy we propose in this paper iterates the specialization process by starting from the derived program $P2$, and propagates the constraints of the error configuration (not those of the initial configuration, as it has been done in our second specialization above). We perform one more specialization, starting from program $P2_{rev}$ obtained by reversing program $P2$ as we now indicate (the general technique will be presented in the next section).

First, program $P2$ can be viewed as a program of the form:

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

if we define the predicates a, trans, and b as follows (round parentheses make a single argument out of a tuple of arguments):

s4. $\mathtt{a((new2,X1,Y1,N))\ :\!-\ N>0,\ X1=1,\ Y1=1.}$
s5. $\mathtt{trans((new2,X,Y,N),(new3,X1,Y1,N))\!:\!-X=1,Y=1,N>1,}$
$\mathtt{X1=2,Y1=3.}$
s6. $\mathtt{trans((new3,X,Y,N),(new3,X1,Y1,N))\!:\!-X1\geq 1,Y1\geq X1,}$
$\mathtt{X1\leq N,\ X1=X+1,\ Y1=X1+Y.}$
s7. $\mathtt{b((new3,X,Y,N))\ :\!-\ Y\geq 1,\ N>0,\ X\geq N,\ X>Y.}$

Indeed, $P2$ can be obtained from s1–s7 by (i) unfolding clauses s1–s3 with respect to $\mathtt{a(U)}$, $\mathtt{trans(U,V)}$, and $\mathtt{b(U)}$, and then (ii) rewriting the atoms of the form $\mathtt{r1((new2,X,Y,N))}$ and $\mathtt{r1((new3,X,Y,N))}$ as $\mathtt{new2(X,Y,N)}$ and $\mathtt{new3(X,Y,N)}$, respectively. (The occurrences of the predicate symbols new2 and new3 in the arguments of r1 should be considered as individual constants.)

Then, the reversed program $P2_{rev}$ is given by the following clauses:

r1. $\mathtt{unsafe\ :\!-\ b(U),\ r2(U).}$
r2. $\mathtt{r2(V)\quad :\!-\ trans(U,V),\ r2(U).}$
r3. $\mathtt{r2(U)\quad :\!-\ a(U).}$

together with clauses s4–s7. For our proof of program safety, the correctness of this program reversal which produces program $P2_{rev}$ from program $P2$, is established by the fact that unsafe $\in M(P2)$ iff unsafe $\in M(P2_{rev})$.

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. Program $P2$ checks the reachability of a configuration $c2$ satisfying b from a configuration $c1$ satisfying a, by moving *forward* from $c1$ to $c2$. Program $P2_{rev}$ checks the reachability of $c2$ from $c1$, by moving *backward* from $c2$ to $c1$. Thus, in the case where a and b are predicates that characterize the initial and final configurations, respectively, the reversal transformation derives a program that checks the reachability of an error configuration from an initial configuration by moving *backward* from the error configuration. In particular, in the body of the clause for unsafe in $P2_{rev}$ the constraint $\mathtt{b(U)}$ contains, among others, the constraint $\mathtt{X>Y}$ characterizing the error configuration and, by specializing $P2_{rev}$, we will propagate the constraints of the error configuration.

**Third Specialization: Propagation of the constraints of the error configuration**

Let us then specialize program $P2_{rev}$. We start from the clauses r1–r3 and s4–s7. We unfold the clause for unsafe (clause r1) with respect to the leftmost atom $\mathtt{b(U)}$ and we get:

32. $\mathtt{unsafe\ :\!-\ Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ r2((new3,X,Y,N)).}$

In order to fold clause 32 we introduce the definition:

33. $\mathtt{new4(X,Y,N)\ :\!-\ Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ r2((new3,X,Y,N)).}$

We fold clause 32, thereby getting:

32.f $\mathtt{unsafe\ :\!-\ Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ new4(X,Y,N).}$

Then we unfold the last definition we have introduced (clause 33) and we get:

34. $\mathtt{new4(X,Y,N)\ :\!-\ Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ a((new3,X,Y,N)).}$
35. $\mathtt{new4(X1,Y1,N)\ :\!-\ Y1\geq 1,\ N>0,\ X1\geq N,\ X1>Y1,}$
$\mathtt{trans(U,(new3,X1,Y1,N)),\ r2(U).}$

By unfolding, clause 34 is deleted because the head of clause s4 is not unifiable with $\mathtt{a(new3,X,Y,N)}$, and by unfolding clause 35 with respect to $\mathtt{trans(U,(new3,X1,Y1,N))}$ we get two clauses each of which has an unsatisfiable constraint in its body. Thus, we are left with clause 32.f only. Since clause 32.f is not a constrained fact, its least model is empty, and thus unsafe does not hold and we may conclude our program verification by stating that the given program is safe with respect to $\varphi_{init}$ and $\varphi_{error}$.

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. It can be shown that, if we perform our specializations by taking into account only the constraints of the initial configuration *or* only the constraints of the error configuration, it is *not* possible to show program safety in our example. Thus, as advocated in this paper, if we perform a sequence of program specializations, we may gain an extra power when we have to prove program properties. This is confirmed by the experiments we have performed on various examples taken from the literature. We will report on those experiments in Section 6.

In the next section we will formally present our method for iterated specialization, where the propagation of the constraints of the initial and the error configurations can be alternated in any order.

## 5. Iterated Specialization

The strategy for iterated specialization we propose in this paper for directing the specialization steps, is depicted in Figure 1.

```
Input:  A CLP program $I$ encoding an unsafety triple.
Output: Program $I_{sp}$ such that unsafe $\in M(I)$ iff unsafe $\in M(I_{sp})$.
$Specialize_{Remove}(I, I_{sp})$;
while there exists a clause in $I_{sp}$ of the form: unsafe :- G, where
G is not the empty goal do
  either $I_{rev} := I_{sp}$ or $Reverse(I_{sp}, I_{rev})$;
  $Specialize_{Prop}(I_{rev}, I_{sp})$;
end-while
```

**Figure 1.** The Iterated Specialization strategy.

The Iterated Specialization strategy takes as input the CLP program $I$ which encodes an unsafety triple as shown is Section 3. Thus, given the unsafety triple $\{\!\{\varphi_{init}\}\!\}\ P\ \{\!\{\varphi_{error}\}\!\}$, program $I$ is made out of: (i) the CLP facts associated with the Core CIL program fragment $P$, (ii) the clauses for the interpreter tr that encodes the transition relation $\Longrightarrow$, (iii) the clauses for the predicates unsafe and reach (see clauses 5–7 of Section 3), (iv) the clauses for initConf and errorConf encoding the formulas $\varphi_{init}$ and $\varphi_{error}$, respectively.

The input program $I$ is specialized by applying the procedure *Specialize*$_{Remove}$ which implements the removal of the interpreter as illustrated in the example of Section 4. As shown in that example, *Specialize*$_{Remove}$ unfolds away the relation tr and introduces new predicate definitions corresponding to (some of the) program points of the original Core CIL program.

Then the strategy iterates the two procedures *Reverse* (which may be skipped) and *Specialize*$_{Prop}$ and, if it terminates, it derives a specialized program which 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 Core CIL program is unsafe, while in the latter case the unsafety property does not hold and the given Core CIL program is safe.

**The Specialize Procedure**

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

Assume that the program $I$ taken as input by the *Specialize* procedure contains $j \geq 1$ clauses defining the predicate unsafe:

$$\mathtt{unsafe\ :\!-\ c_1(X),p_1(X),\ \ \ldots,\ \ unsafe\ :\!-\ c_j(X),p_j(X)}$$

where $\mathtt{c_1(X),\ldots,c_j(X)}$ are either atoms or constraints, and $\mathtt{p_1(X),\ldots,p_j(X)}$ are atoms. For instance, when $I$ is the initial version of the interpreter for the subset of CIL considered in this paper, we have that $j$ is 1, $c_1$(X) is initConf(X), and $p_1$(X) is reach(X).

The *Specialize* procedure modifies the initial program $I$ by propagating the information encoded by $c_1$(X), ..., $c_j$(X), which characterize the initial or the error configurations, depending on the number of applications of the *Reverse* procedure. In particular, by unfolding we may be able to discover that unsafe has a successful derivation, and hence the given Core CIL program fragment is unsafe. Alternatively, by unfolding we may add constraints that are inconsistent with the ones occurring in the constrained facts, and by folding we may derive mutually recursive predicates, and hence these predicates will have no constrained facts and we infer safety.

The *Specialize* procedure makes use of two functions: *Unf* and *Gen*, for controlling unfolding and generalization, respectively.

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 $\{\mathtt{K}_i$ :- $\mathtt{c}_i$, $\mathtt{B}_i \mid i = 1, \ldots, m\}$ be the set of the (renamed apart) clauses in program $I$ such that, for $i = 1, \ldots, m$, A is unifiable with $\mathtt{K}_i$ via the most general unifier $\vartheta_i$. We define the following function:

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

Each clause in $\mathit{Unf}(C, \mathtt{A})$ is said to be derived by *unfolding* $C$ *w.r.t.* A. In order to perform unfolding during specialization, we assume that 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 refer to [23] for a survey of techniques for controlling unfolding that guarantee this finiteness property.

The *Specialize* procedure makes use of the function *Gen*, called *generalization operator* for introducing new predicate definitions. Given a clause $E$: newp(X) :- e(X,X1), p(X1) and the set *Defs* of clauses that define the new predicates introduced up to a given point by the specialization algorithm, $\mathit{Gen}(E, \mathit{Defs})$ returns a clause $G$: newr(X) :- g(X), p(X) such that: (i) newr is a new predicate symbol, and (ii) e(X, X1) $\sqsubseteq$ g(X1), where $\sqsubseteq$ denotes *entailment* between constraints. Then, $E$ is *folded* by using $G$, thereby deriving the new clause newp(X) :- e(X,X1), newr(X1). By the correctness of the folding rule this transformation step preserves equivalence with respect to the least model semantics. Indeed, newr(X1) is equivalent to the conjunction g(X1),p(X1) by definition and the conjunction e(X,X1),g(X1) is equivalent to e(X,X1).

The generalization operator used in the *Specialize*$_{\mathit{Remove}}$ procedure returns a clause $G$: newr(X) :-p(X) (that is, g(X) is the constraint true). The generalization operators used in the *Specialize*$_{\mathit{Prop}}$ procedure is based on *widening*, *convex hull*, and *well-quasi orderings* relations which have been introduced for analyzing and specializing constraint logic programs [8, 12, 27]. For lack of space we do not present here the definitions of the generalization operators, and for more details the reader may refer to [12], where it is also shown that these operators guarantee that during specialization only a finite number of new predicates is introduced.

In the *Specialize* procedure we also use the following notions. A clause of the form H :- c, B is *subsumed* by the constrained fact H :- d if c $\sqsubseteq$ d. We say that a predicate p in a program $P$ is *useless* if for all predicates q such that p depends on q there is no constrained fact for q in $P$, where the *dependency relation* between predicates is defined as usual.

## 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 [10].

```
Input: Program $I$.
Output: Program $I_{sp}$ such that unsafe $\in M(I)$ iff unsafe $\in M(I_{sp})$.
INITIALIZATION:
$I_{sp} := \emptyset$;
$InCls := \{$unsafe:- $c_1$(X),$p_1$(X), ..., unsafe:- $c_j$(X),$p_j$(X)$\}$;
$Defs := \emptyset$;

while in InCls there is a clause $C$ which is not a constrained fact do
  UNFOLDING:
  $SpC := Unf(C, A)$, where $A$ is the leftmost atom in the body of $C$;
  while in SpC there is a clause $D$ whose body contains an occurrence of a unfoldable atom $A$ do
      $SpC := (SpC - \{D\}) \cup Unf(D, A)$
  end-while;

  CLAUSE REMOVAL:
  while in SpC there are two distinct clauses $E$ and $F$ such that $E$
      subsumes $F$ or there is a clause $F$ whose body contains an
      unsatisfiable constraint do
      $SpC := SpC - \{F\}$
  end-while;

  DEFINITION-INTRODUCTION & FOLDING:
  while in SpC there is a clause $E$ of the form:
      H :- e(X,X1),p(X1)
  where H is either unsafe or an atom of the form newp(X), and
  e(X,X1) is a constraint do
    if in Defs there is a clause $D$ of the form:
        newq(X) :- c(X), p(X)
      where c(X) is a constraint such that e(X,X1) $\sqsubseteq$ c(X1)
    then $SpC := (SpC - \{E\}) \cup \{$H :- e(X,X1), newq(X1)$\}$;
    else let $Gen(E, Defs)$ be newr(X) :- g(X), p(X)
        where: (i) newr is a predicate symbol not occurring
        in $I \cup Defs$, and (ii) g(X) is a constraint such that
        e(X,X1) $\sqsubseteq$ g(X1);
        $Defs := Defs \cup \{Gen(E, Defs)\}$;
        $InCls := InCls \cup \{Gen(E, Defs)\}$;
        $SpC := (SpC - \{E\}) \cup \{$H :- e(X,X1), newr(X1)$\}$
  end-while;

  $InCls := InCls - \{C\}$;
  $I_{sp} := I_{sp} \cup SpC$;
end-while;

REMOVAL OF USELESS CLAUSES:
Remove from $I_{sp}$ all clauses whose head predicate is useless.
```

**Figure 2.** The *Specialize* Procedure.

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 defined as one of the generalization operators presented in [12], a *finite* set of new predicates are introduced during the *Specialize* procedure, and hence the 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 on the input program* $I$. *Then* unsafe $\in M(I)$ *iff* unsafe $\in M(I_{sp})$.

## The Reverse Transformation

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

The output of the *Specialize* procedure, and hence the input of the *Reverse* procedure, is a program $I_{sp}$ of the form:

```
unsafe :- a1(X), newp1(X).
    ...
unsafe :- ak(X), newpk(X).
newq1(X) :- t1(X,X1), newr1(X1).
    ...
newqm(X) :- tm(X,X1), newrm(X1).
news1(X) :- b1(X).
    ...
newsn(X) :- bn(X).
```

where: (i) $\mathtt{a}_1(\mathtt{X}), \ldots, \mathtt{a}_k(\mathtt{X}), \mathtt{t}_1(\mathtt{X},\mathtt{X1}), \ldots, \mathtt{t}_m(\mathtt{X},\mathtt{X1}), \mathtt{b}_1(\mathtt{X}), \ldots, \mathtt{b}_n(\mathtt{X})$ are constraints, and (ii) the (possibly non-distinct) predicate symbols $\mathtt{newp}_i$'s, $\mathtt{newq}_i$'s, $\mathtt{newr}_i$'s, and $\mathtt{news}_i$'s are the new predicate symbols introduced by the generalization operator *Gen*.

The *Reverse* procedure transforms program $I_{sp}$ in two steps as follows.
*Step* 1. Program $I_{sp}$ is transformed into a program $I'_{sp}$ of the following form:

```
unsafe :- a(U), r1(U).
r1(U) :- trans(U,V), r1(V).
r1(U) :- b(U).
a((newp1,X)) :- a1(X).
    ...
a((newpk,X)) :- ak(X).
trans((newq1,X),(newr1,X1)) :- t1(X,X1).
    ...
trans((newqm,X),(newrm,X1)) :- tm(X,X1).
b((news1,X)) :- b1(X).
    ...
b((newsn,X)) :- bn(X).
```

The correctness of the transformation from $I_{sp}$ to $I'_{sp}$ relies on the fact that by unfolding the clauses of $I'_{sp}$ w.r.t. $\mathtt{a}(\mathtt{U})$, $\mathtt{trans}(\mathtt{U},\mathtt{V})$, and $\mathtt{b}(\mathtt{U})$, and then rewriting all atoms of the form $\mathtt{r1}((\mathtt{newpred},\mathtt{Z}))$ into $\mathtt{newpred}(\mathtt{Z})$, we get back $I_{sp}$.
*Step* 2. Program $I'_{sp}$ is transformed into a program $I_{rev}$ by replacing the first three clauses of $I'_{sp}$ by the following ones:

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

The correctness of this transformation can be proved as indicated in [3], and thus we have the following result.

THEOREM 3. *Let $I_{rev}$ be the program derived from program $I_{sp}$ by the Reverse procedure. Then* $\mathtt{unsafe} \in M(I_{sp})$ *iff* $\mathtt{unsafe} \in M(I_{rev})$.

Finally, by using Theorems 1, 2, and 3, we get the following soundness result.

THEOREM 4. (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 $I_{sp}$ is the output of the strategy, then $P$ is safe with respect to $\varphi_{init}$ and $\varphi_{error}$ iff* $\mathtt{unsafe} \notin I_{sp}$.

## 6. Experimental Evaluation

We have performed an experimental evaluation of our software model checking method on 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.

Programs *substring* and *tracerP* are taken from [21] and [17], respectively, while programs $re1$ and *singleLoop* are taken from [9]. Program *selectSort* is an encoding of the selection sort algorithm where references to arrays have been replaced by using uninitialized variables to perform array bounds checking. The other programs are taken from the benchmark set of DAGGER [14]. The source code of all the programs we have considered is available at `http://map.uniroma2.it/smc/`.

Our software model checker consists of three modules.
(i) A front-end module, based on CIL [26], which translates 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 which is used for removing the interpreter and applying the iterated specialization strategy. This module is implemented using the MAP system [24], which is a tool for transforming constraint logic programs written in SICStus Prolog. The MAP system operates on constraints over the rational numbers by using the `clpq` library.
(iii) A module for inspecting the CLP programs obtained by specialization and checking whether they contain the fact `unsafe` (in which case the given C programs are proved unsafe) or they contain no clauses with head `unsafe` (in which case the given C program are proved safe).

We have also tested the following three state-of-the-art CLP-based software model checkers for C programs: (i) ARMC [29], (ii) HSF(C) [13], and (iii) TRACER [18]. ARMC and HSF(C) are based on the Counter-Example Guided Abstraction Refinement technique (CEGAR) [4, 20, 31], while TRACER uses a technique based on approximated preconditions and approximated postconditions. We have compared the performance of those model checkers on our benchmark programs with that of our model checker.

Table 1 reports the results of our experimental evaluation, which has been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

In the columns labelled with MAP(a) and MAP(b) we have reported the time needed for the verification process using the MAP system according to the method for iterated specialization presented in this paper. That time includes the time needed for removing the interpreter, which ranges from some tenths of milliseconds to eight or nine seconds, for the most complex programs.

We have used the following sequences of program transformations (the exponent indicates the number of times the associated subsequence has been applied):
for MAP(a):
*Specialize*$_{Remove}$; (*Reverse*; *Specialize*$_{Prop}$)$^n$, and
for MAP(b):
*Specialize*$_{Remove}$; *Specialize*$_{Prop}$; (*Reverse*; *Specialize*$_{Prop}$)$^{n-1}$.
Thus, after the removal of the interpreter, the first *Specialize*$_{Prop}$ of MAP(a) propagates the constraints of the *error configuration* (and this corresponds to a *backward* move along the transition graph associated with the reachability relation), while the first *Specialize*$_{Prop}$ of MAP(b) propagates the constraints of the *initial configuration* (and this corresponds to a *forward* move along the transition graph).

In the subcolumns labelled with $n$ we have reported the total number of program specializations needed, after the removal of the interpreter, before a successful verification or before timeout.

| Program to be verified | MAP(a) | | MAP(b) | | ARMC | HSF(C) | TRACER | |
|---|---|---|---|---|---|---|---|---|
| | $n$ | | $n$ | | | | *SPost* | *WPre* |
| *barber* | 3 | 137.99 | 2 | 69.74 | 577.06 | 13.88 | 12.86 | 3.86 |
| *barber*1 | 1 | 13.71 | 2 | 26.43 | 414.01 | 0.59 | 7.00 | 5.17 |
| *berkeleyNat* | 3 | 1.88 | 2 | 1.51 | 11.48 | 0.29 | - | 1.33 |
| *berkeley* | 1 | 1.57 | 2 | 1.53 | 11.28 | 0.26 | - | 1.00 |
| *efm* | 3 | 6.48 | 2 | 4.04 | 31.17 | 0.51 | 2.43 | 2.68 |
| *ex*1 | 1 | 0.03 | 2 | 0.40 | 1.69 | 0.22 | - | 1.39 |
| $f1a$ | 2 | 0.17 | 1 | 0.07 | - | 0.21 | - | 1.97 |
| *heapSort* | 1 | 8.16 | 2 | 13.51 | 39.66 | 0.35 | - | - |
| *heapSort*1 | 1 | 3.01 | 2 | 9.58 | 20.55 | 0.26 | - | - |
| *interp* | 1 | 0.12 | 2 | 0.28 | 11.41 | 0.19 | - | 2.92 |
| *lifnat* | 3 | 23.13 | 2 | 20.20 | 228.96 | 7.19 | - | 72.12 |
| *lifo* | 1 | 20.56 | 2 | 15.59 | 126.54 | 0.54 | - | 7.45 |
| $p2$ | 1 | 14.75 | 1 | - | - | 0.77 | - | - |
| *re*1 | 1 | 0.23 | 1 | 0.08 | - | 0.19 | - | - |
| *seesaw* | 1 | 2.09 | 2 | 3.04 | - | 0.27 | - | 34.16 |
| *selectSort* | 3 | 1.96 | 6 | 3.26 | 24.97 | 0.25 | - | - |
| *singleLoop* | 3 | 0.35 | 2 | 0.28 | - | - | - | 56.57 |
| *substring* | 2 | 0.16 | 1 | 0.20 | 472.32 | 40.51 | - | - |
| *swim* | 3 | 116.56 | 2 | 40.13 | - | 2.94 | - | 15.13 |
| *tracerP* | 1 | 0.01 | 1 | 0.07 | - | - | 1.04 | 1.03 |
| number of verified programs | | 20 (9) | | 19 (15) | 13 | 18 | 4 | 14 |
| total time | | 353.29 | | 209.94 | 1971.10 | 69.42 | 23.33 | 206.78 |

**Table 1.** Time (in seconds) required for program verification. '-' means 'unable to verify within 10 minutes'. The subcolumns labelled by $n$ report the number of *Specialize*$_{Prop}$ performed by the MAP system, that is, the number of specializations, after the removal of the interpreter.

The generalizations performed by the MAP system are done by applying the widening and the convex hull operators.

In the remaining columns we have reported the results obtained by ARMC, HSF(C), and TRACER using the strongest postcondition (*SPost*) and the weakest precondition (*WPre*) options, respectively. The last two lines report the number of programs which have been successfully verified and the total time needed for verification. For the MAP system we also report, between parentheses, the number of programs which require more than one iteration ($n > 1$) to be successfully verified. This number measures the effectiveness of performing additional iterations of program specialization.

The experimental results show that our approach of iterating program specialization, is indeed effective and determines an increase of the number of successful verifications. Sometimes the increase is substantial.

On our set of examples, the MAP(a) system is able to verify 20 programs out of 20. It is followed by MAP(b) (19), HSF(C) (18), TRACER using weakest precondition (14), ARMC (13), and TRACER using strongest postcondition (4).

We observe that some of the examples are verified by the MAP system using the transformation sequence MAP(a) with $n = 1$, that is, by propagating the constraints of the error configuration only. The examples *re*1 and *tracerP* can be verified by MAP(a) or MAP(b) with $n = 1$, that is, by a single propagation of the constraints of either the error configuration or the initial configuration, respectively.

Thus, in some of our benchmark programs the invariants which are useful for their proofs, can be discovered by a *single propagation* of the constraints of either the error configuration or the initial configuration. However, if we perform a preliminary additional specialization by propagating the constraints of the other configuration (that is, the initial configuration or the final configuration, respectively), then we are still able to prove the property of interest by requiring very little extra time. Actullay, sometimes (see, for instance, the *lifo* and *substring* programs) this additional preliminary specialization can even reduce the total time because it prunes the search space.

Looking at Table 1, we may conclude that the verification times taken by our MAP-based software model checker is generally comparable with that of the other tools, and it is not much greater than that of the fastest tools.

## 7. 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 [9]. The main novelties introduced in this work are the following: (i) we consider CIL programs (that is, C programs transformed by using the CIL tool [26]), and (ii) we define a general verification framework in which specialization of constraint logic programs is repeatedly applied with the objective of making a more effective use of the information dispersed through the program to be verified (typically, the initial configurations and the error configurations).

The use of constraint logic programming and program specialization for the verification of properties of imperative programs is not novel. It has also been investigated, for instance, in [28]. 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 analyser for CLP programs is applied to the residual program for computing 'invariants' of the input imperative program, which are used in the proof of the properties of interest. Unlike [28], our verification approach does not require any static analyzer and, instead, we discover program invariants during the specialization process by means of suitable generalization operators. They are defined in terms of operators and relations on constraints such as widening, convex-hull, and well-quasi orders [12]. As in [28], we also use program specialization to perform the removal of the interpreter, but in addition, we repeatedly use specialization for propagating the information about the initial configurations and the error configurations.

A popular technique for program verification is the so called Counter-Example Guided Abstraction Refinement (CEGAR) [4, 20, 31], which is used by many software model checkers such as BLAST [2], DAGGER [14], 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 form 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 performed using that refined abstraction in the hope of a successful proof.

In the field of static program analysis the idea of performing backward and forward semantic analyses has been proposed in [6]. These analyses have been combined, for instance, in [7], 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 that algorithm, which improves the abstract state space exploration and makes use of disjunctive abstract domain, has been proposed in [30].

Our approach can be regarded as complementary to those 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 plays a role similar to that of abstraction. (Note, however, that 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.

Our preliminary experimental results show that our approach is viable and competitive with state-of-the-art software model checkers, some of which follow the CEGAR approach.

As a future work, we would like to address the issue of the design of suitable heuristics that should guide the choice of the generalization operators in each iteration of the specialization process. It will also be important to design suitable strategies for controlling the number of clauses of the specialized programs. Indeed, if that number is too high, then verification becomes inefficient.

## 8. Acknowledgments

We thank the anonymous referees for their constructive comments.

## References

[1] T. Ball and V. Levin and S. K. Rajamani. A decade of software model checking with SLAM. Commun. ACM, vol. 54, no. 7, 68–76, 2011.

[2] D. Beyer, T.A. Henzinger, R. Jhala and R. Majumdar. The software model checker Blast: Applications to software engineering, *Int. J. Softw. Tools Technol. Transf.*, vol. 9, no. 5, 505–525. Springer, 2007.

[3] D. R. Brough and C. J. Hogger. Grammar-Related Transformations of Logic Programs. *New Generation Computing*, 9 (1), 115–134, 1991.

[4] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In: *Proc. CAV'00*, 154–169. Springer, 2000.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: *Proc. POPL'77*, 238–252. ACM Press, 1977.

[6] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In: *Proc. POPL'79*, 269–282. ACM Press, 1979.

[7] P. Cousot, R. Ganty, and J.-F. Raskin. Fixpoint-Guided Abstraction Refinements. In: *Proc. SAS'07*, LNCS 4634, 333–348. Springer, 2007.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In: *Proc. POPL'78*, 84–96. ACM Press, 1978.

[9] E. De Angelis, F. Fioravanti, A. Pettorossi, and M. Proietti. Branching Preserving Specialization for Software Model Checking. In: *Preliminary Proc. LOPSTR'12*, E. Albert, ed., Report CW 625, Katholieke Universiteit Leuven, Belgium, 28–44, 2012.

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

[11] F. Fioravanti, A. Pettorossi, and M. Proietti, Automated strategies for specializing constraint logic programs. In: *Proc. LOPSTR'00*, LNCS 2042, 125–146. Springer, 2001.

[12] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization Strategies for the Verification of Infinite State Systems. *Theory and Practice of Logic Programming*, 2012.

[13] S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In: *Proc. TACAS'12*. To appear, 2012.

[14] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In: *Proc. TACAS'08*, LNCS 4963, 443–458. Springer, 2008.

[15] N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11:157–185, 1997.

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

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

[18] J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification, 2012.

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

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

[21] R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In: *Proc. TACAS'06*, LNCS 3920, 459–473. Springer, 2006.

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

[23] 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.

[24] The MAP transformation system. `www.iasi.cnr.it/∼proietti /system.html`

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

[26] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC'02*, LNCS 2304, 209–265. Springer, 2002.

[27] J. C. Peralta, J. P. Gallagher. Convex Hull Abstractions in Specialization of CLP Programs. In: *Proc. LOPSTR'02*, LNCS 2664, 90–108. Springer, 2003.

[28] J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In: *Proc. SAS'98*, LNCS 1503, 246–261. Springer, 1998.

[29] A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: *Proc. PADL'07*, LNCS 4354, 245–259. Springer, 2007.

[30] F. Ranzato, O. Rossi-Doria, and F. Tapparo. A forward-backward abstraction refinement algorithm. In *Proc. VMCAI'08*, LNCS 4905, 248–262. Springer, 2008.

[31] H. Saïdi. Model checking guided abstraction and analysis. In *Proc. SAS'00*, LNCS 1824, 377–396. Springer, 2000.