Software Model Checking by Program Specialization

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

[1] University 'G. D'Annunzio',
Viale Pindaro 42, I-65127 Pescara, Italy
{deangelis,fioravanti}@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present a method for performing model checking of imperative programs by using techniques based on the specialization of constraint logic programs (CLP). We have considered a simple imperative language, called SIMP, extended with a nondeterministic choice operator, and we have introduced a CLP interpreter which defines the operational semantics of SIMP. Our software model checking method which consists in: (1) translating a given SIMP program, together with the safety property to be verified and a description of the input values, into terms, (2) specializing the CLP interpreter with respect to the above translation, and (3) computing the least model of the specialized interpreter. By inspecting the derived least model we can verify whether or not the given SIMP program satisfies the safety property. The method is fully automatic and has been implemented using the MAP transformation system. We have shown the effectiveness of our method by applying it to some examples taken from the literature and we have compared its performance with that of other state-of-the-art software model checkers.

1 Introduction

Formal verification of software products has recently received a growing attention as a promising methodology for increasing the reliability and reducing the cost of software production (e.g. by reducing the time to market).

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see our discussion in Section 5 and [12] for a recent survey).

Among the various software model checking techniques, there are some which make use of *Constraint Logic Programming* (CLP). This programming paradigm has been shown to be very suitable for the symbolic evaluation and the analysis of imperative programs (see, for instance, [10,11,15,16]).

Also the technique we present in this paper makes use of CLP and addresses the problem of verifying *safety* properties of imperative programs. Basically, a safety property states that an unsafe configuration (or an error configuration) is not reachable from an initial configuration by any execution of the program. Since we consider programs that act on integer numbers, the problem of deciding whether or not an unsafe configuration is reachable is undecidable.

In previous work we have shown that the termination of reachability analyses of infinite state systems can be improved by encoding reachability as a CLP program and then specializing the CLP program by incorporating the information about the initial and the unsafe states [4].

In this paper we show that the approach presented in [4] can be extended to perform software model checking of SIMP programs, that is, simple imperative programs which also include a nondeterministic choice operator. In order to make this extension we follow an approach similar to the one proposed in [15] and, in particular, we introduce a CLP program which encodes an interpreter defining the operational semantics of imperative programs.

Then, in order to verify a safety property of a given SIMP program c, we introduce a predicate `unsafe` which holds if and only if an unsafe configuration is reachable from an initial configuration by any execution of the CLP interpreter taking c as input. Thus, we can show that program c is safe by checking that `unsafe` does not belong to the least model of the CLP interpreter with c as the input program.

Unfortunately, it is often the case that the construction of that least model does not terminate. In order to mitigate this problem, we specialize the CLP interpreter with respect to program c and the property characterizing its input values. We show through experiments that the computation of the least model of the specialized program terminates in many interesting cases. Since program specialization preserves the least model of CLP programs, we can verify whether or not the given SIMP program c satisfies a given safety property by inspecting the least model of the specialized CLP interpreter and checking whether or not it contains `unsafe`.

Our software model checking method consists of the following three steps. First, Step (1): we translate the given SIMP program, together with the safety property to be verified and a description of the input values, into terms, then, Step (2): we specialize the CLP interpreter with respect to the terms derived at the end of the previous step, and finally, Step (3): we compute the least model of the specialized interpreter. By inspecting the derived least model we can verify whether or not the given SIMP program satisfies the safety property.

We have implemented our software model checking method using the MAP transformation system [14] and we have shown that our method is competitive with state-of-the-art software model checking systems such as ARMC [16] and TRACER [8].

The paper is organized as follows. In Section 2 we describe the syntax of the SIMP language and the CLP interpreter which defines its operational semantics. In Section 3 we describe our software model checking approach and the particu-

lar CLP program specialization technique which we use. In Section 4 we report on some experiments we have performed by using a prototype implementation based on the MAP transformation system. We also compare the results we have obtained using the MAP system with the results we have obtained using ARMC and TRACER. Finally, in Section 5 we discuss the related work and, in particular, we compare our approach with other existing software model checking methods.

2 A CLP Interpreter for a Simple Imperative Language

In this section we describe the syntax and the semantics of a simple imperative language (SIMP) which is based on the IMP language [18] and extends it with the nondeterministic choice operator *ndc* (often denoted by $*$ in the literature).

We also introduce a CLP encoding of an interpreter for SIMP which defines the structural operational semantics (also known as small-step semantics) of the SIMP language, and by means of an example, we show how to translate SIMP programs to CLP terms.

The syntax of SIMP is built upon the following sets:

- *Int* of integer constants, ranged over by the variable n,
- *Bool* of boolean constants {**true**, **false**},
- *Loc* of locations, ranged over by the variable x,
- *AExpr* of arithmetic expressions, ranged over by the variable a,
- *BExpr* of boolean expressions, ranged over by the variable b,
- *TExpr* of test expressions, ranged over by the variable t, and
- *Com* of commands, ranged over by the variable c.

The abstract syntax of the language is as follows.

$a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2$

$b ::= \mathbf{true} \mid \mathbf{false} \mid a_1 \ op \ a_2 \mid !\, b \mid b_1 \ \&\& \ b_2 \mid b_1 \ \mid\mid \ b_2$

$t ::= ndc \mid b$

$c ::= skip \mid x = a \mid c_1; c_2 \mid \mathbf{if}\ t\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid \mathbf{while}\ t\ \mathbf{do}\ c\ \mathbf{od}$

where $op \in \{<, >, <=, >=, ==, !=\}$ and the operators !, &&, and || define 'not', 'and' and 'or', respectively.

In order to define the operational semantics of SIMP commands we need the following notions. An *environment* e is a finite function from *Loc* to *Int*. We write $e[n/x]$ to denote the environment e' such that $e'(x) = n$ and $e'(y) = e(y)$ for every $y \neq x$. A *configuration* is a pair $\langle c, e\rangle$ of a command c and an environment e. The operational semantics is defined in terms of a transition relation $\Longrightarrow$ over configurations. We say that a configuration $\langle c', e'\rangle$ is *reachable* from the configuration $\langle c, e\rangle$ if $\langle c, e\rangle \Longrightarrow^* \langle c', e'\rangle$.

Here are the axioms and inference rules defining the operational semantics of commands:

$$\frac{}{\langle ndc,\, e\rangle \longrightarrow \mathbf{true}} \qquad \frac{}{\langle ndc,\, e\rangle \longrightarrow \mathbf{false}}$$

$$\frac{\langle a, e\rangle \longrightarrow n}{\langle x = a\,;\, c,\, e\rangle \implies \langle c, e[n/x]\rangle}$$

$$\frac{}{\langle skip\,;\, c,\, e\rangle \implies \langle c, e\rangle}$$

$$\frac{}{\langle ((c_1\,;\, c_2)\,;\, c_3)\,,\, e\rangle \implies \langle (c_1\,;\, (c_2\,;\, c_3)\,,\, e\rangle}$$

$$\frac{\langle t, e\rangle \longrightarrow \mathbf{true}}{\langle (\mathbf{if}\, t\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2)\,;\, c,\, e\rangle \implies \langle c_1\,;\, c, e\rangle}$$

$$\frac{\langle t, e\rangle \longrightarrow \mathbf{false}}{\langle (\mathbf{if}\, t\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2)\,;\, c,\, e\rangle \implies \langle c_2\,;\, c, e\rangle}$$

$$\frac{}{\langle \mathbf{while}\, t\, \mathbf{do}\, c_1\, \mathbf{od}\,;\, c,\, e\rangle \implies \langle (\mathbf{if}\, t\, \mathbf{then}\, (c_1\,;\, \mathbf{while}\, t\, \mathbf{do}\, c_1\, \mathbf{od})\, \mathbf{else}\, \mathit{skip})\,;\, c,\, e\rangle}$$

together with the usual rules which define the relation $\longrightarrow$ for the operational semantics of arithmetic and boolean expressions (see, for instance, [18]).

Note that, similarly to what is done in [15], we have defined the operational semantics of commands under the assumption that every command is either *skip* or a concatenation of commands ending by *skip*. In the next section we present the CLP program which encodes these rules for the operational semantics of commands.

A (*software model checking*) *specification* is a triple of the form $\langle$*init-prop*, *com*, *unsafe-prop*$\rangle$, where *com* is a command, and *init-prop* and *unsafe-prop* are boolean expressions. An environment e is said to be *initial* (or *unsafe*) if the boolean expression *init-prop* (or *unsafe-prop*, respectively) is true in e. A configuration $\langle c, e\rangle$ is said to be *initial* (or *unsafe*) if the environment e is initial (or unsafe, respectively).

We say that a command *com* satisfies the specification $\langle$*init-prop*, *com*, *unsafe-prop*$\rangle$ (or *com* is safe, for short) if no unsafe configuration is reachable from an initial configuration of the form $\langle com, e\rangle$, for some environment e.

2.1 A CLP Interpreter for SIMP Commands

Now we introduce a CLP program which encodes the operational semantics of the SIMP language. We assume that the reader is familiar with the basic notions of constraint logic programming [7].

An environment is a list of terms each of which is of the form `lv(loc(`v`),V)`, where `V` is a variable which stores the value associated with the location v. We assume that the locations used in our programs are known in advance and, thus,

the lists used to represent the environments have a fixed length. We also introduce two auxiliary predicates `lookup` and `update`, that operate on environments. The predicate `lookup` is defined as follows:

```
lookup(loc(X), [lv(loc(X),Y)|_], V) :- V=Y.
lookup(loc(X), [_|T], V) :- lookup(loc(X), T, V).
```

thus, `lookup(loc(X),E,V)` holds iff `V` is the value associated with the location `X` in the environment `E`. The predicate `update` is defined as follows:

```
update(loc(X), Y, [lv(loc(X),_)|T], [lv(loc(X),V)|T]) :- V=Y.
update(loc(X), V, [H|T], [H|T1]) :- update(loc(X), V, T, T1).
```

thus, `update(loc(X),V,E1,E2)` holds iff `E2` is the environment equal to `E1` except that `V` is the value associated with the location `X`.

Now, we introduce the predicate `aeval` which defines the semantics function for arithmetic expressions. The predicate `aeval(A,E,V)` holds iff $\langle \mathtt{A}, \mathtt{E} \rangle \longrightarrow \mathtt{V}$, that is, `V` is the value of the arithmetic expression `A` in the environment `E`.

```
aeval(int(N),_,V)       :- V=N.
aeval(loc(X),E,V)       :- V=N,     lookup(loc(X),E,N).
aeval(plus(A1,A2),E,V)  :- V=V1+V2, aeval(A1,E,V1), aeval(A2,E,V2).
aeval(minus(A1,A2),E,V) :- V=V1-V2, aeval(A1,E,V1), aeval(A2,E,V2).
aeval(mult(A1,A2),E,V)  :- V=V1*V2, aeval(A1,E,V1), aeval(A2,E,V2).
```

The semantics function for boolean expressions is encoded by using the predicate `beval`, which is shown below. The predicate `beval(B,E)` holds iff $\langle \mathtt{B}, \mathtt{E} \rangle \longrightarrow \mathbf{true}$, that is, the boolean expression `B` evaluates to **true** in the environment `E`.

```
beval(true,_).
beval(eq(A1,A2),E)       :- T1=T2,   aeval(A1,E,T1), aeval(A2,E,T2).
beval(lte(A1,A2),E)      :- T1=<T2,  aeval(A1,E,T1), aeval(A2,E,T2).
beval(lt(A1,A2),E)       :- T1<T2,   aeval(A1,E,T1), aeval(A2,E,T2).
beval(gte(A1,A2),E)      :- T1>=T2,  aeval(A1,E,T1), aeval(A2,E,T2).
bexpr(gt(A1,A2),E)       :- T1>T2,   aeval(A1,E,T1), aeval(A2,E,T2).
beval(or(B1,_),E)        :- beval(B1,E).
beval(or(_,B2),E)        :- beval(B2,E).
beval(and(B1,B2),E)      :- beval(B1,E), beval(B2,E).
beval(neq(A1,A2),E)      :- beval(not(eq(A1,A2)),E).

beval(not(eq(A1,A2)),E)  :- beval(or(lt(A1,A2),gt(A1,A2)),E).
beval(not(lte(A1,A2)),E) :- beval(gt(A1,A2),E).
beval(not(lt(A1,A2)),E)  :- beval(gte(A1,A2),E).
beval(not(gte(A1,A2)),E) :- beval(lt(A1,A2),E).
beval(not(gt(A1,A2)),E)  :- beval(lte(A1,A2),E).
beval(not(or(B1,B2)),E)  :- beval(and(not(B1),not(B2)),E).
beval(not(and(B1,B2)),E) :- beval(or(not(B1),not(B2)),E).
beval(not(neq(A1,A2)),E) :- beval(eq(A1,A2),E).
```

Notice that the evaluation of arithmetic and boolean expressions does not modify the environment.

In order to define the CLP clauses for the semantics of commands we introduce the term `s(c,e)` encoding a configuration $\langle c, e \rangle$, where `c` is a command

and `e` is an environment. A command is encoded as a term built out of the following constructors: `skip` for the empty command, `asgn` for assignment, `comp` for command composition, `ite` for **if-then-else**, and `while` for **while-do**.

The transition relation $\Longrightarrow$ over configurations is encoded by the predicate `t` which is defined by the clauses we give below.

The predicate `t( s(C,E), s(C1,E1) )` holds iff `s(C,E)` $\Longrightarrow$ `s(C1,E1)`, that is, the execution of the command `C` in the environment `E` leads to the execution of the command `C1` in the environment `E1`.

```
t( s(comp(skip,S),E), s(S,E) ).
t( s(comp(asgn(loc(X),A),S),E1), s(S,E2) ) :-
              aeval(A,E1,V), update(loc(X),V,E1,E2).
t( s(comp(comp(S1,S2),S3),E),   s(comp(S1,comp(S2,S3) ),E)).
t( s(comp(ite(B,S1,_),S3),E),   s(comp(S1,S3),E) ) :- beval(B,E).
t( s(comp(ite(B,_,S2),S3),E),   s(comp(S2,S3),E) ) :- beval(not(B),E).
t( s(comp(ite(ndc,S1,_),S3),E), s(comp(S1,S3),E) ).
t( s(comp(ite(ndc,_,S2),S3),E), s(comp(S2,S3),E) ).
t( s(comp(while(B,S1),S2),E),
              s(comp(ite(B,comp(S1,while(B,S1)),skip),S2),E) ).
```

We have not written the clauses for the semantics of the test expression *ndc* because we have implicitly considered them in the **if-then-else** command.

Now, we introduce a CLP program Bw which, by using a bottom-up evaluation strategy, performs the reachability analysis over configurations in a backward way, starting from unsafe configurations.

Definition 1 (Encoding Program). Given a specification $\langle$*init-prop*, *com*, *unsafe-prop*$\rangle$, the program Bw consists of the following clauses:

```
unsafe :- init(X), bwReach(X).
bwReach(X) :- unsafe(X).
bwReach(X) :- t(X,X1), bwReach(X1).
init(s(com,E)) :- beval(init-prop,E).
unsafe(s(_,E)) :- beval(unsafe-prop,E).
```

together with the clauses for the predicates `aeval`, `beval`, and `t`. In program Bw, `com`, `init-prop`, and `unsafe-prop` stand for the terms which are the result of the translation of the command *com* and the boolean expressions *init-prop* and *unsafe-prop*, respectively.

The predicate `bwReach(X)` holds iff an unsafe configuration is reachable from the configuration `X` (that is, `X` is backward reachable from some unsafe configuration) and the predicate `unsafe` holds iff there is an initial configuration `X` of the form `s(com,E)` such that `bwReach(X)` holds. Thus, in order to verify that `com` is safe, it is sufficient to show that `unsafe` cannot be derived by using program Bw.

In the following example, we show how to translate commands and test expressions and derive the corresponding terms. This example is the program *tracer_prog_d* taken from [10], which has also been used in the experimental evaluation as reported in Section 4.

Example 1. Let us consider a specification $\langle$*init-prop*, *com*, *unsafe-prop*$\rangle$ where:

- *init-prop* is `x==0 && y>=0 && error==0`
- *com* is the command:

```
while ( x < 10000) {
    y = y + 1;
    x = x + 1;
}
if ( y + x < 10000)
    error = 1;
```

- *unsafe-prop* is `error==1`

The components *init-prop*, *com*, and *unsafe-prop* of that specification are translated as follows:

- *init-prop* is translated into

```
and(eq(loc(x),int(0)),and(eq(loc(y),int(0)),eq(loc(error),int(0))))
```

- *com* is translated into

```
comp(comp(while(lt(loc(x),int(10000)),
                comp(asgn(loc(y),plus(loc(y),int(1))),
                     asgn(loc(x),plus(loc(x),int(1))))),
          ite(lt(plus(loc(y),loc(x)),int(10000)),
              asgn(loc(error),int(1)),
              skip)),
     skip)
```

- *unsafe-prop* is translated into `eq(loc(error),int(1))`

The environment is a term of the form:

```
[lv(loc(y),Y),lv(loc(x),X),lv(loc(error),Error)].
```

The following theorem establishes the correctness of the translation from a given specification S to the CLP program Bw. The proof of this theorem follows immediately from the correctness of the clauses for `t` (which encode the operational semantics of commands) and the correctness of the backward reachability algorithm.

Theorem 1 (Correctness of Encoding). *Let S=$\langle$init-prop, com, unsafe-prop$\rangle$ be a specification, Bw be the CLP program obtained from S as indicated in Definition* 1*, and $M(Bw)$ be the least model of Bw. Then, the command com satisfies the specification S iff* `unsafe` $\notin M(Bw)$.

3 Software Model Checking by Specialization of CLP Programs

Let $S = \langle$*init-prop*, *com*, *unsafe-prop*$\rangle$ be a specification. Our method for verifying whether or not *com* is safe consists of three steps.

Step (1). We translate the given specification into the terms `init-prop`, `com`, and `unsafe-prop` as indicated in the previous section.

Step (2). We specialize the program Bw with respect to the command `com` and the boolean expression `init-prop` that characterizes the set of initial environments. The output of this step is the program $SpBw$ such that `unsafe` $\in M(Bw)$ iff `unsafe` $\in M(SpBw)$.

Step (3). We compute the least model $M(SpBw)$ of the specialized program $SpBw$ and we infer that `com` is safe iff `unsafe` $\notin M(SpBw)$.

The objective of Step (2) is to modify the initial program Bw by propagating the information specified by the term `com` encoding the command, and the term `init-prop` characterizing the set of the initial environments. By exploiting this information, the computation of the least model $M(SpBw)$ may be more effective and terminate more often than the computation of the least model $M(Bw)$. In particular, the interpretation overhead is compiled away by specialization, thereby producing a CLP program where the term encoding the command is no longer present. Also the boolean expression that characterizes the initial environments is propagated through the structure of the specialized CLP program.

Step (2) is realized by a specialization algorithm adapted from [4]. This algorithm makes use of the following transformation rules: *definition introduction*, *unfolding*, *clause removal*, and *folding* which, under suitable conditions, guarantee that the least model semantics is preserved by specialization (see, for instance, [3]).

The specialization starts from the clause `unsafe :- init(X), bwReach(X)` and iterates the application of two procedures: (i) the *Unfold* procedure, which applies the *unfolding* and *clause removal* rules, and (ii) the *Generalize&Fold* procedure, which applies the *definition introduction* and *folding* rules.

The *Unfold* procedure takes as input a clause γ of the form `H :- c(X), bwReach(X)` and returns as output a set Γ of clauses derived from γ by *one* or more applications of the unfolding rule, which consists in replacing an atom A occurring in the body of a clause by the bodies of the clauses in Bw whose head is unifiable with A. The first step of the *Unfold* procedure consists in unfolding γ with respect to `bwReach(X)`. The subsequent steps are performed according to the following strategy.

A clause is unfolded with respect to an atom in its body if and only if that atom is either of the form `init`(...), or `unsafe`(...), or `aeval`(...), or `beval`(...), or `t`(...), or `lookup`(...), or `update`(...), or `bwReach(s(c,e))`, where `c` is a ground term *not* of the form `comp(while(b,s1),s2)`.

Due to the structure of the clauses in Bw, the *Unfold* procedure terminates for every input clause γ. Note that, in particular, in order to enforce termination, no atom of the form `bwReach(s(comp(while(b,s1),s2),e),t2)` is selected for unfolding after the first unfolding step, thereby avoiding a potentially infinite unrolling of **while-do** loops.

At the end of the *Unfold* procedure, clauses with unsatisfiable constraints and subsumed clauses are removed.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and introduces a set *NewDefs* of *definitions*, that is,

clauses of the form `newp(X) :- d(X), bwReach(X)`, where `newp` is a new predicate symbol corresponding to specialized versions of the `bwReach` predicate. Any such definition denotes a set of configurations `X` satisfying the constraint `d(X)`. By folding the clauses in Γ using the definitions in *NewDefs* and the definitions introduced at previous iterations of the specialization algorithm, the procedure derives a new set of specialized clauses. In particular, a clause of the form:

```
newq(X) :- c(X), bwReach(X)
```

obtained by the *Unfold* procedure is folded by using a definition of the form:

```
newp(X) :- d(X), bwReach(X)
```

if `c(X)` entails `d(X)`, denoted `c(X)` $\sqsubseteq$ `d(X)`. The result of folding is the specialized clause:

```
newq(X) :- c(X), newp(X).
```

The specialization algorithm proceeds by applying the *Unfold* procedure followed by the *Generalize&Fold* procedure to each clause in *NewDefs*, and terminates when no new definitions are needed for applying folding steps.

Unfortunately, an uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions, thereby causing the nontermination of the specialization algorithm. In order to guarantee termination, the *Generalize&Fold* procedure may introduce new definitions which are *more general than* definitions introduced by previous applications of the procedure, where the *more general than* relation between definitions is as follows: a definition:

```
newr(X) :- g(X), bwReach(X)
```

is more general than the definition

```
newp(X) :- d(X), bwReach(X)
```

if `d(X)` $\sqsubseteq$ `g(X)`. Thus, more general definitions correspond to larger sets of configurations.

The generalization operator we use in our experiments, reported in Section 4, is defined in terms of relations and operators on constraints such as *widening*, *convex-hull*, and *well-quasi orders*. We will not describe in detail the generalization operator we apply, and we refer to [4,5,15] for various operators which can be used for specializing constraint logic programs. It will be enough to say that the termination of the specialization algorithm is ensured by the fact that, similarly to the widening operator presented in [2], our generalization operator guarantees that during specialization only a finite number of new predicates is introduced.

Since the correctness of the specialization algorithm directly follows from the fact that the transformation rules preserve the least model semantics [3], we have the following result.

Theorem 2 (Termination and Correctness of Specialization). (i) *The specialization algorithm terminates.* (ii) *Let program SpBw be the output of the specialization algorithm. Then* `unsafe` $\in M(Bw)$ *iff* `unsafe` $\in M(SpBw)$.

In order to compute the least model of *SpBw* as required by Step (3), we apply a procedure called *BottomUp*. This procedure computes sets of atoms represented as sets of *constrained facts*, that is, sets of (possibly non-ground) clauses of the form `H :- c`, where `H` is an atom and `c` is a constraint. The least model $M(SpBw)$ is constructed by computing the least fixpoint of the *non-ground immediate consequence operator* S_{SpBw}, instead of the usual immediate consequence operator T_{SpBw} [7]. Since this fixpoint may consist of an infinite set of constrained facts, the *BottomUp* procedure may not terminate. In Section 4 we will see that the *BottomUp* procedure, applied after the specialization algorithm, terminates in our examples.

Example 2. The following program *SpBw* is obtained as output of the specialization algorithm when it takes as input the CLP program of Example 1:

```
new3(X,Y,E) :- X>=0, X<10000, Y>=X, E=0, X1=X+1, Y1=Y+1, new3(X1,Y1,E).
new3(X,Y,E) :- X>=10000, Y>=X, E=0, new5(X,Y,E).
new2(X,Y,E) :- X=0, Y>=0, E=0, X1=1, Y1=Y+1, new3(X1,Y1,E).
unsafe      :- X=0, Y>=0, E=0, new2(X,Y,E).
```

Notice that the second clause for `new3` can be removed from the program because it contains a call to a predicate `new5` whose definition is empty. Since the program contains no constrained fact, we have that $M(SpBw)$ is empty, and thus the original specification is safe.

4 Experimental Evaluation

In this section we present some preliminary results concerning our software model checking technique which we have applied to several examples taken from the literature.

We have realized our model checker as a software tool which consists of three modules: (i) a translator from SIMP specifications to terms, (ii) the MAP system for program specialization, and (iii) a CLP program for computing the least models of CLP programs. The MAP system [14] is a tool for transforming constraint logic programs implemented in SICStus Prolog which uses the `clpr` library to operate on constraints over the reals.

We have performed the model checking of the following C programs: (i) $f1a$, (ii) $f2$, (iii) *seesaw*, (iv) $JM06$, (v) *prog_dagger*, (vi) *tracer_prog_d*, (vii) *interpolants_needed*, and (viii) *widen_needed*. The source codes of the above C programs, which can be given as input to the MAP system, are available at `http://map.uniroma2.it/smc/`.

Programs $f1a$, $f2$, *seesaw*, and $JM06$ are benchmarks which have been used to evaluate the performance of DAGGER. Unfortunately, we were not able to compare our results with those which can be obtained by DAGGER because of its unavailability. Nevertheless, the source code of the benchmark programs is available at `http://www.cfdvs.iitb.ac.in/~bhargav/dagger.php`. In particular, $JM06$ is the benchmark program in [13]. Programs *prog_dagger*, *interpolants_needed*, and *widen_needed* have been taken from [6] (and the related technical report) and *tracer_prog_d* from [10].

All experiments have been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB under the GNU Linux operating system. The results of our experiments are shown in Table 1.

Table 1. Time (in seconds) taken for performing model checking. $\perp$ denotes 'terminating with error' (TRACER, using the default options, terminates with 'Fatal Error: Heap overflow'). ∞ means 'Model checking not successful within 20 minutes'.

Programs	ARMC	TRACER	MAP
$f1a$	∞	$\perp$	0.08
$f2$	∞	$\perp$	7.58
$JM06$	719.39	180.09	10.20
prog_dagger	∞	$\perp$	5.37
seesaw	3.41	$\perp$	0.03
tracer_prog_d	∞	0.01	0.03
interpolants_needed	0.13	$\perp$	0.06
widen_needed	∞	$\perp$	0.07

We have also performed our experiments on the same set of examples by using two state-of-the-art software model checkers which use CLP: (i) ARMC [16], and (ii) TRACER [8], and we have compared the results obtained by using those tools and our software model checker based on the MAP system.

ARMC is a CLP-based software model checker which realizes the *Counter Example Guided Abstraction Refinement* (CEGAR) technique (see Section 5 for a detailed description). ARMC relies on a CIL front-end (C Intermediate language, `http://cil.sourceforge.net/`), called `c2armc`, which translates a subset of C, annotated with error labels, into a transition system whose transitions are constraints over variables occurring in the C program.

TRACER is a Symbolic Execution (SE)-based model checker for the verification of safety properties of sequential C programs. It uses a CIL front-end to translate a C program, annotated with safety properties, into a CLP program, and uses a symbolic interpreter to execute the CLP translation to prove that unsafe states are unreachable.

TRACER, using the default options, fails in all examples in which a nondeterministic while-do, possibly composed with nondeterministic if-then-else statements, is used ($f1a$, $f2$, *prog_dagger*, *seesaw*, *interpolants_needed*, and *widen_needed*). In these cases, the nondeterministic choice is obtained by using a function which returns an unknown integer. Indeed, even if TRACER provides effective improvements when using symbolic execution for the verification of programs with unbounded execution of loops, it fails (that is, it terminates with 'Fatal Error: Heap overflow') when the unbounded number of executions is a value returned by a function. Also ARMC is unable to prove within 20

minutes these examples, with the exception of the program *seesaw* and *interpolants_needed*.

ARMC is unable to prove *tracer_prog_d* safe within 20 minutes. In fact, in order to prove that *tracer_prog_d* is safe, a CEGAR-based approach has to unroll the while-do loop (it requires at least 10000 iterations of the CEGAR loop to discover the right set of predicates). TRACER, conversely, succeeds in 0.01 seconds. Indeed, this example has been used by the TRACER designer to illustrate the benefits of SE-based approached with respect to CEGAR-based approaches.

Both ARMC and TRACER succeed with $JM06$, but TRACER performs much better than ARMC.

Now let us compare the model checking times obtained by using ARMC and TRACER with those obtained with our method, which are reported in column called MAP of Table 1. This last column shows the total model checking time including both the specialization time and the least model computation time. Our method succeeds where both ARMC and TRACER fail ($f1a$, $f2$, *prog_dagger*, and *widen_needed*). Instead, in $JM06$, where both ARMC and TRACER succeed, our method substantially reduces the total time of verification. In the analysis of programs *seesaw* and *interpolants_needed* we perform better than ARMC, whereas in *tracer_prog_d* our method has a slight decrease of performance, but this is not discouraging because, indeed, the program *tracer_prog_d* has been designed to illustrate a situation where the techniques used by TRACER work well.

5 Related Work and Conclusions

We have proposed a model checking method, based on the specialization of CLP programs, to verify safety properties of a simple imperative language extended with a nondeterministic choice operator, called SIMP. The method is fully automatic and has been implemented using the MAP transformation system.

In particular, we have defined an interpreter for the operational semantics of the SIMP language and we have encoded it as a CLP program. Then, given a specification $\langle$*init-prop*, *com*, *unsafe-prop*$\rangle$, where *init-prop* and *unsafe-prop* are boolean expressions describing the initial and the unsafe environments, respectively, and *com* is the command to be executed; the MAP transformation system has been used to specialize the interpreter with respect to *com* and *init-prop*. Finally, by computing the least model of the specialized CLP program we verify whether or not the SIMP command *com* in the initial environment *init-prop* is safe, that is, none of the reachable configurations satisfied *unsafe-prop*. Note that our verification method can be applied by using a forward reachability algorithm, instead of the backward reachability algorithm, as we have done here.

We have applied our approach to some C programs taken from the literature and we have compared the results in terms of efficiency and precision with two state-of-art software model checkers: ARMC and TRACER. Table 1 shows that our model checker is competitive with the considered software model checkers.

Our software model checking approach is an extension of the one presented in [4,5] for the verification of safety properties of infinite state reactive systems. In that work the authors encode an infinite state reactive system as a CLP program and then they apply program specialization to improve the effectiveness of the reachability analysis.

The use of constraint logic programs to verify properties of simple imperative programs has also been proposed by [15]. That paper introduces a general framework in which well-known techniques developed to analyse declarative programs, that is, partial evaluation and static analysis of constraint logic programs, are exploited to analyse imperative programs. In particular, the formal semantics of an imperative language is encoded into a constraint logic program. Then, the interpreter is partially evaluated with respect to a specific imperative program to obtain a residual program on which a static analyser for CLP programs is applied. Finally, the information gathered during this process is translated back to the original imperative program. The correctness of the analysis follows from the correctness of the partial evaluator and the static analyser for the declarative language.

Our approach does not require static analysis of CLP and, instead, we use a bottom-up evaluator to compute the least model of the CLP program obtained by program specialization. Moreover, in our approach we specialize the given program also with respect to the property characterizing its input values.

A widely used verification technique implemented by software model checkers (e.g. SLAM, BLAST and ARMC) is the *CounterExample Guided Abstraction Refinement* [12,17]. The key component of the CEGAR technique is a loop which, given a program and a safety property, repeatedly checks whether or not the abstract model of that program is safe and, based on the counterexamples possibly found, refines the abstract model until, hopefully, the program is proved to be safe.

In particular, the CEGAR loop starts with an initial, coarse grained, abstraction of the program. If the abstraction is proved to be safe then the original program is guaranteed to be safe, otherwise a counterexample (that is, an execution which makes the program unsafe) is produced. The counterexample is then analysed in the concrete program: if it turns out to be a *genuine* counterexample (that is, an execution which can be reproduced by the concrete program), then the program is proved to be unsafe, otherwise it is a *spurious* counterexample which has been generated due to a too coarse abstraction. The abstraction is, thus, refined to remove that counterexample, and the verification loop continues.

In a completely different manner from CEGAR-based software model checkers, our tool starts with a CLP program which models the most detailed model of the program and the abstraction is obtained as a side effect of the generalization operators used to force termination of the specialization process.

The CEGAR approach has also been implemented by using CLP programs. In particular, in [16] the authors have designed a CLP-based model checker for C programs with an abstraction refinement, called ARMC. The abstract model of the program to be verified is obtained by using a predicate abstraction

domain, that is, a set of constraints whose free variables are program variables. The set of reachable states are, thus, computed in the abstract model and if none of the unsafe states is generated then the program is proved to be safe. The CEGAR loop starts with an empty set of constraints and in each iteration extends it by using a solver which generates interpolants for linear arithmetic constraints with uninterpreted function symbols.

A different technique is that of symbolic execution which has been first used for program testing and recently has been applied to program verification [10]. Symbolic execution (SE) of programs is a method which uses a symbolic representation of inputs, instead of actual inputs, to execute them. Thus, the outcome of statements are formulas over the symbolic representation of the input values. Program execution is modelled by using a symbolic execution tree whose paths are symbolic representations of program computations.

In [8], the authors have designed a SE-based software model checker, called TRACER, for the verification of finite-state safety properties of C programs. TRACER, differently from CEGAR-based tools, starts with a finer grained model of the program and uses abstraction refinement to make symbolic execution to be effective in practice when the program to be verified consists of unbounded loops. In particular, TRACER tries to construct a finite symbolic execution tree which overapproximates the set of all concrete reachable states. In doing this, the abstraction refinement removes all paths which are irrelevant to prove that the unsafe state are reachable by computing interpolants.

Our preliminary experimental results show that our approach is viable and competitive with state-of-the-art software model checkers. Thus, we think that our approach can be effective in practice. In order to prove this claim, in the near future, we plan to do experiments on a larger set of examples and compare our results with other state-of-the-art software model checker to get better insights about the differences with respect to related approaches. We also plan to extend our interpreter to deal with more sophisticated features of imperative languages such as arrays, pointers, and procedure calls.

We plan to investigate how to integrate information about the unsafe property during the generalization process to obtain a verification method which is closer to the use of the counterexamples in the CEGAR approach and in the Min-max algorithm [10].

Finally, we would like to explore in more detail the way how program specialization can be effectively used to improve the precision and the efficiency of some existing state-of-the-art software model checkers. In particular, we plan to verify whether a post-processing of the specialized program can be given as input to those software model checkers.

References

1. T. Ball, A. Podelski, and S. Rajamani. Relative Completeness of Abstraction Refinement for Software Model Checking. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 2280, pages 158–172. Springer Berlin, 2002.

2. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the Fifth ACM Symposium on Principles of Programming Languages, POPL '78*, pages 84–96. ACM Press, 1978.
3. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
4. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Improving Reachability Analysis of Infinite State Systems by Specialization. In *Proceedings of the 5th international conference on Reachability Problems, RP '11*, pages 165–179. Springer Berlin, 2011.
5. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theory and Practice of Logic Programming.* Available on CJO 2012 doi:10.1017/S1471068411000627. Special Issue on the 25th GULP Annual Conference, 2012.
6. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS* '08, pages 443–458, 2008. (Also available as Technical Report CFDVS, IIT Bombay, TR-07-23)
7. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
8. J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification. `http://www.clip.dia.fi.upm.es/~jorge/tracer/`.
9. J. Jaffar, J. A. Navas, and A. E. Santosa. Unbounded Symbolic Execution for Verification. `http://www.comp.nus.edu.sg/~joxan/res.html`
10. J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, arXiv:1103.2027v1, 2011.
11. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In Ian Gent, editor, *Principles and Practice of Constraint Programming, CP '09*, Lecture Notes in Computer Science 5732, pages 454–469. Springer Berlin, 2009.
12. R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.
13. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Proc. 12th Int Conf. TACAS '06, Vienna, Austria, March 25-April 2, 2006, Lecture Notes in Computer Science 3920, pages 459–473. Springer, 2006.
14. The MAP Transformation System. `http://www.iasi.cnr.it/~proietti/system.html`. Also available via the WEB interface `http://www.map.uniroma2.it/mapweb`.
15. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In G. Levi, editor, *Static Analysis, 5th International Symposium, SAS '98*, Pisa, Italy, September 14-16, 1998, Lecture Notes in Computer Science 1503, pages 246–261. Springer, 1998.
16. 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*, Lecture Notes in Computer Science 4354, pages 245–259. Springer Berlin, 2007.
17. N. Sharygina, S. Tonetta, and A. Tsitovich. An abstraction refinement approach combining precise and approximated techniques. *Software Tools for Technology Transfer*, 14(1):1–14, 2012.
18. G. Winskel. *The Formal Semantics of Programming Languages: An Introduction.* The MIT Press, Cambridge, Massachusetts, 1993.