Software Model Checking by Program Specialization

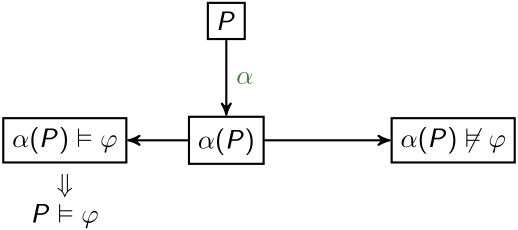
Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2] and Maurizio Proietti[3]

[1]University of Chieti-Pescara 'G. d'Annunzio'
[2]University of Rome 'Tor Vergata'
[3]CNR-IASI, Rome

CILC 2012
Rome, 6 June 2012

Software model checking

- given:
  1. a program P
  2. a formal specification φ of its behaviour
- create a conservative abstraction $\alpha(P)$ of P
- verify whether or not $\alpha(P)$ satisfies φ

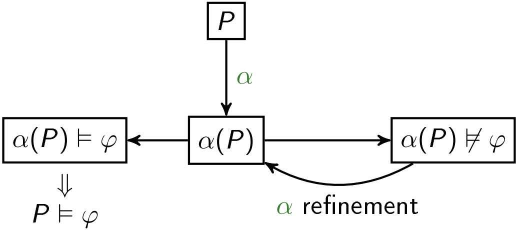


Clarke et al. *CEGAR for Symbolic Model Checking.*

Cousot and Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program.*

Software model checking

- given:
  1. a program P
  2. a formal specification φ of its behaviour
- create a conservative abstraction $\alpha(P)$ of P
- verify whether or not $\alpha(P)$ satisfies φ



Clarke et al. *CEGAR for Symbolic Model Checking*.

Cousot and Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program*.

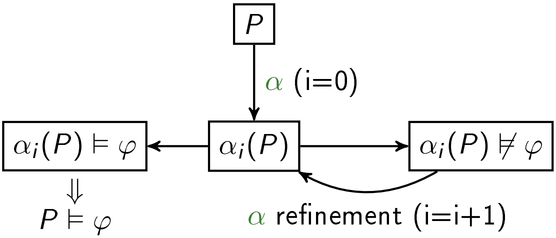
Software model checking

Modelling software

abstraction $\alpha(P)$:

- must be sound: if $\alpha(P) \models \varphi$ then $P \models \varphi$
- should be as precise as possible

$$\alpha_1(P) \sqsubseteq \alpha_2(P) \sqsubseteq \cdots \sqsubseteq \alpha_i(P) \sqsubseteq \cdots$$



Program Specialization

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program specialization is a framework for performing an Agile, Iterative and Evolutionary development of verification techniques and tools:

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program specialization is a framework for performing an Agile, Iterative and Evolutionary development of verification techniques and tools:

- soundness of abstraction

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program specialization is a framework for performing an Agile, Iterative and Evolutionary development of verification techniques and tools:

- soundness of abstraction
- parametricity w.r.t. languages and logics

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program specialization is a framework for performing an Agile, Iterative and Evolutionary development of verification techniques and tools:

- soundness of abstraction
- parametricity w.r.t. languages and logics
- compositionality of program transformations

Program Specialization

Why using program specialization?

Program specialization is a transformation technique whose objective is the adaptation of a program to a context of use.

Program specialization is a framework for performing an Agile, Iterative and Evolutionary development of verification techniques and tools:

- soundness of abstraction
- parametricity w.r.t. languages and logics
- compositionality of program transformations
- modularity separation of language features and verification techniques

Specialization-based Software Model Checking

Verification Framework

Given:

- a program P written in a language L, and
- a property φ in a logic M,

we can verify that φ holds for P by:

Phase 1: writing an interpreter I for L and a semantics S for M in Constraint Logic Programming,

Phase 2: creating a model of P by specializing the interpreter I and the semantics S with respect to P and φ, and

Phase 3: analyzing the specialized program (by, possibly, repeating Phase 2).

Peralta et al. *Analysis of Imperative Programs through Analysis of Constraint Logic Programs.*

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

- R1 Definition
- R2 Unfolding
- R3 Folding
- R4 Clause removal

Etalle and Gabbrielli. *Transformations of CLP modules.*

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding

R3 Folding

R4 Clause removal

Etalle and Gabbrielli. *Transformations of CLP modules.*

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ w.r.t.

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding

R4 Clause removal

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ w.r.t.

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding $p(X_1, \ldots, X_n) \leftarrow c \wedge A$ w.r.t. A by using

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ if $c \Rightarrow d$

R4 Clause removal

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \dots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \dots, X_n) \leftarrow c \wedge q(X_1, \dots, X_n)$ w.r.t.

$q(X_1, \dots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \dots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding $p(X_1, \dots, X_n) \leftarrow c \wedge A$ w.r.t. A by using

$q(X_1, \dots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \dots, X_n) \leftarrow c \wedge q(X_1, \dots, X_n)$ if $c \Rightarrow d$

R4 Clause removal

R4.1 $p(X_1, \dots, X_n) \leftarrow c \wedge q(X_1, \dots, X_n)$

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ w.r.t.

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding $p(X_1, \ldots, X_n) \leftarrow c \wedge A$ w.r.t. A by using

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ if $c \Rightarrow d$

R4 Clause removal

R4.1 ~~$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$~~ if c is unsatisfiable

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ w.r.t.

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding $p(X_1, \ldots, X_n) \leftarrow c \wedge A$ w.r.t. A by using

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ if $c \Rightarrow d$

R4 Clause removal

R4.1 ~~$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$~~ if c is unsatisfiable

R4.2 $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$, $p(X_1, \ldots, X_n) \leftarrow d$

Etalle and Gabbrielli. *Transformations of CLP modules.*

R1 Definition $newp(X_1, \ldots, X_n) \leftarrow c \wedge A$

R2 Unfolding $p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ w.r.t.

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge d \wedge A$

R3 Folding $p(X_1, \ldots, X_n) \leftarrow c \wedge A$ w.r.t. A by using

$q(X_1, \ldots, X_n) \leftarrow d \wedge A$

gives

$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$ if $c \Rightarrow d$

R4 Clause removal

R4.1 ~~$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$~~ if c is unsatisfiable

R4.2 ~~$p(X_1, \ldots, X_n) \leftarrow c \wedge q(X_1, \ldots, X_n)$~~, $p(X_1, \ldots, X_n) \leftarrow d$
if $c \rightarrow d$ (subsumption)

Etalle and Gabbrielli. *Transformations of CLP modules.*

Software model checking

Specialization strategy

Spec(Π,c) **begin**

$\Pi_{Sp} = \emptyset$;

$Def = \{c\}$;

while $\exists q \in Def$ **do**

Unf = Clause Removal(Unfold(q));

$Def = Def - \{q\} \cup Define(Unf)$;

$\Pi_{Sp} = \Pi_{Sp} \cup Fold(Unf, Def)$

done

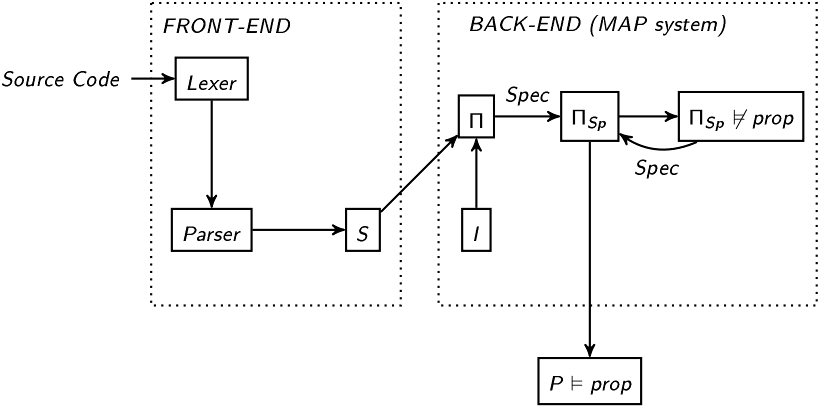
end

Theorem: $\Pi \models \varphi$ iff $\Pi_{Sp} \models \varphi$

- Generalizations in *Define*($\cdot$) ensure termination of *Spec*, but may prevent the proof of the property.

Software model checking

Framework Architecture



P and φ are encoded as S and *prop*, respectively.

Specialization-based Software Model Checking

Verification Framework: SIMP language and safety properties

$$
\begin{array}{lcl}
a & ::= & n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \\
b & ::= & \textbf{true} \mid \textbf{false} \mid a_1 \ op \ a_2 \mid \, !\, b \mid b_1 \ \&\& \ b_2 \mid b_1 \ \| \ b_2 \\
t & ::= & ndc \mid b \\
c & ::= & skip \mid x = a \mid c_1 ; c_2 \mid \textbf{if} \ t \ \textbf{then} \ c_1 \ \textbf{else} \ c_2 \mid \textbf{while} \ t \ \textbf{do} \ c \ \textbf{od}
\end{array}
$$

CLP interpreter for the operational semantics of SIMP

```
tr(s(skip,S), E).
tr(s(asgn(var(X),A),E),s(skip,E1)) :- aeval(A,S,V), update(var(X),V,S,E1).
tr(s(comp(C0,C1),S), s(C1,S1)) :- tr(s(C0,S),S1).
tr(s(comp(C0,C1),S), s(comp(C0',C1),S')) :- tr(s(C0,S), s(C0',S')).
tr(s(ite(B,C0,_),S), s(C0,S)) :- beval(B,S).
tr(s(ite(B,_,C1),S), s(C1,S)) :- beval(not(B),S).
tr(s(ite(ndc,S1,_),E),s(S1,E)).
tr(s(ite(ndc,_,S2),E),s(S3,E)).
tr(s(while(B,C),S), s(ite(B,comp(C,while(B,C)),skip),S)).
```

Specialization-based Software Model Checking

Verification Framework: SIMP language and safety properties

Let P be a SIMP program and φ be a safety property.

- Phase 1: Encode P and φ into a CLP program Π

```
reachable(X) :- unsafe(X).
reachable(X) :- tr(X,X'), reachable(X').
unsafe :- initial(X), reachable(X).
unsafe(s(error,E)).
initial(s(T,E)) :- init_constraint(E).
```

  where:

  - `tr(X,X')` encodes the operational semantics I of SIMP.
  - `s(T,E)` encodes P (instructions `T` and variables `E`)

- Phase 2: *Spec* - Specialize Π w.r.t.

```
initial(s(P,E)) :- init_constraint(E).
```

- Phase 3: *BuEval* - Bottom up Evaluation of Π_{Sp}

P is safe iff $unsafe \notin BuEval(\Pi)$ iff $unsafe \notin BuEval(\Pi_{Sp})$.

Example

Phase 1: Encoding of P and φ

```
int x=0; int y=0; int n;
assume(n>0);
while (x<n) { x = x+1; y = y+1; }
if (y>x) error;
```

↓

```
1. initial(
     s(comp(while(lt(var(x),var(n)),
                comp(asgn(var(x),plus(var(x),int(1))),
                     asgn(var(y),plus(var(y),int(1))))),
                ite(gt(var(y),var(x)),error,skip)),
        [lv(x,X),lv(y,Y),lv(n,N)])) :- X=0,Y=0,N>0.
2. unsafe(s(error,_)).
```

Example

Phase 2: Specialization of I w.r.t. *P*

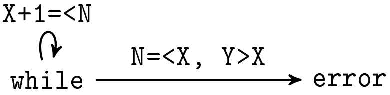
1. `initial(s(comp(while(··),···),[lv(x,X),···])) :- X=0,···,N>0.`
2. `unsafe(s(error,_)).`

+

3. CLP Interpreter

↓

```
new1(X,Y,N) :- X+1=<N, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, Y>X.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```



Example

Phase 3: Bottom Up Evaluation of the Specialized Program

Let Π_{Sp} the specialized CLP program:

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, Y>=X+1.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

BuEval(Π_{Sp}) = {

```
  new1(X,Y,N) :- X+1=<Y, N=<X.
  new1(X,Y,N) :- X+1=<Y, N=X+1.
  new1(X,Y,N) :- X+1=<Y, N=X+2.
  new1(X,Y,N) :- X+1=<Y, N=X+3.
  new1(X,Y,N) :- X+1=<Y, N=X+4.
  .... }
```

Example

Phase 3: Bottom Up Evaluation of the Specialized Program

Let Π_{Sp} the specialized CLP program:

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, Y>=X+1.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

BuEval(Π_{Sp}) = {

```
  new1(X,Y,N) :- X+1=<Y, N=<X.
  new1(X,Y,N) :- X+1=<Y, N=X+1.
  new1(X,Y,N) :- X+1=<Y, N=X+2.
  new1(X,Y,N) :- X+1=<Y, N=X+3.
  new1(X,Y,N) :- X+1=<Y, N=X+4.
  .... }
```

The Bottom Up Evaluation does not terminate.

Example

Phase 3: Bottom Up Evaluation of the Specialized Program

Let Π_{Sp} the specialized CLP program:

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, Y>=X+1.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

BuEval(Π_{Sp}) = {

```
  new1(X,Y,N) :- X+1=<Y, N=<X.
  new1(X,Y,N) :- X+1=<Y, N=X+1.
  new1(X,Y,N) :- X+1=<Y, N=X+2.
  new1(X,Y,N) :- X+1=<Y, N=X+3.
  new1(X,Y,N) :- X+1=<Y, N=X+4.
  .... }
```

The Bottom Up Evaluation does not terminate.
Thus, we are not able to prove, or disprove, the safety of the given imperative program!

Example

Phase 2: Specialization of Π_{Sp}

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, X+1=<Y.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

↓

```
new2(X,Y,N) :- N>=X,X'=X+1,Y'=Y+1,X'>=Y',Y'>=1,new2(X',Y',N).
new1(X,Y,N) :- X=0, Y=0, N>=1, Y'=1, X'=1, new2(X',Y',N).
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

Example

Phase 2: Specialization of Π_{Sp}

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, X+1=<Y.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

↓

```
new2(X,Y,N) :- N>=X,X'=X+1,Y'=Y+1,X'>=Y',Y'>=1,new2(X',Y',N).
new1(X,Y,N) :- X=0, Y=0, N>=1, Y'=1, X'=1, new2(X',Y',N).
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

No facts

Example

Phase 2: Specialization of Π_{Sp}

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, X+1=<Y.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

↓

```
new2(X,Y,N) :- N>=X, X'=X+1, Y'=Y+1, X'>=Y', Y'>=1, new2(X',Y',N).
new1(X,Y,N) :- X=0, Y=0, N>=1, Y'=1, X'=1, new2(X',Y',N).
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

No facts

The Bottom Up Evaluation terminates

Example

Phase 2: Specialization of Π_{Sp}

```
new1(X,Y,N) :- N>=X+1, X'=X+1, Y'=Y+1, new1(X',Y',N).
new1(X,Y,N) :- N=<X, X+1=<Y.
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

↓

```
new2(X,Y,N) :- N>=X,X'=X+1,Y'=Y+1,X'>=Y',Y'>=1,new2(X',Y',N).
new1(X,Y,N) :- X=0, Y=0, N>=1, Y'=1, X'=1, new2(X',Y',N).
unsafe :- X=0, Y=0, N>=1, new1(X,Y,N).
safe :- not unsafe.
```

No facts
The Bottom Up Evaluation terminates
Thus, the given imperative program is proved to be safe!

Experiments

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
Substring	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

Jaffar et al. *TRACER: A Symbolic Execution Tool for Verification.*

Podelski and Rybalchenko. *ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement.*

Conclusions

- Program specialization is a suitable framework for defining verification procedures which are parametric w.r.t. the languages of
  - the program, and
  - the property

  to be verified
- Preliminary results show that this approach is also viable in practice and competitive with other CLP-based software model checkers
- We are extending the verification framework with
  - more sophisticated language features of imperative language (e.g., pointers, function calls);
  - different properties (e.g., content-sensitive properties)