Software Model Checking
by Program Specialization

Emanuele De Angelis¹, Fabio Fioravanti¹, Alberto Pettorossi² and Maurizio Proietti³

¹University of Chieti-Pescara ‘G. d’Annunzio’
²University of Rome ‘Tor Vergata’
³CNR-IASI, Rome

CILC 2012
Rome, 6 June 2012
Software model checking

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

\[
P \models \varphi \Downarrow \alpha(P) \models \varphi \not\models \varphi
\]

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 $\varphi$ of its behaviour
- create a conservative abstraction $\alpha(P)$ of $P$
- verify whether or not $\alpha(P)$ satisfies $\varphi$

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

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

\[\begin{array}{c}
\alpha_i(P) \models \varphi \\
P \models \varphi
\end{array}\]  \[\begin{array}{c}
\alpha_i(P) \not\models \varphi \\
\alpha \text{ refinement (i=i+1)}
\end{array}\]
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 \( \varphi \) in a logic \( M \),

we can verify that \( \varphi \) 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 \( \varphi \), 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

Specialization-based Software Model Checking
Rules for Specializing CLP Programs

R1 Definition \( \text{newp}(X_1, \ldots, X_n) \leftarrow c \land A \)

R2 Unfolding

R3 Folding

R4 Clause removal

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

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

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

\[ q(X_1, \ldots, X_n) \leftarrow d \land A \]

gives

\[ p(X_1, \ldots, X_n) \leftarrow c \land d \land A \]

R3 Folding

R4 Clause removal

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition \( \text{newp}(X_1, \ldots, X_n) \leftarrow c \land A \)

R2 Unfolding \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \) w.r.t. \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land d \land A \)

R3 Folding \( p(X_1, \ldots, X_n) \leftarrow c \land A \) w.r.t. \( A \) by using \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \) if \( c \Rightarrow d \)

R4 Clause removal

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition \( \text{new} p(X_1, \ldots, X_n) \leftarrow c \land A \)

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

\( q(X_1, \ldots, X_n) \leftarrow d \land A \)

gives

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

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

\( q(X_1, \ldots, X_n) \leftarrow d \land A \)

gives

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

R4 Clause removal

R4.1 \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition \( \text{new}p(X_1, \ldots, X_n) \leftarrow c \land A \)

R2 Unfolding \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \) w.r.t. \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\[ p(X_1, \ldots, X_n) \leftarrow c \land d \land A \]

R3 Folding \( p(X_1, \ldots, X_n) \leftarrow c \land A \) w.r.t. \( A \) by using
\[ q(X_1, \ldots, X_n) \leftarrow d \land A \]
gives
\[ p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \quad \text{if } c \Rightarrow d \]

R4 Clause removal

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

Specialization-based Software Model Checking

Rules for Specializing CLP Programs

R1 Definition \( \text{newp}(X_1, \ldots, X_n) \leftarrow c \land A \)

R2 Unfolding \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \) w.r.t. \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land d \land A \)

R3 Folding \( p(X_1, \ldots, X_n) \leftarrow c \land A \) w.r.t. \( A \) by using \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)
if \( c \Rightarrow d \)

R4 Clause removal

R4.1 \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)
if \( c \) is unsatisfiable
R4.2 \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n), p(X_1, \ldots, X_n) \leftarrow d \)

Specialization-based Software Model Checking
Rules for Specializing CLP Programs

R1 Definition  \( \text{newp}(X_1, \ldots, X_n) \leftarrow c \land A \)

R2 Unfolding  \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \) w.r.t.
\( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land d \land A \)

R3 Folding  \( p(X_1, \ldots, X_n) \leftarrow c \land A \) w.r.t. \( A \) by using
\( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)  \( \text{if} \ c \Rightarrow d \)

R4 Clause removal

R4.1  \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)  \( \text{if} \ c \) is unsatisfiable
R4.2  \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n), p(X_1, \ldots, X_n) \leftarrow d \)
if \( c \rightarrow d \) (subsumption)

Software model checking

Specialization strategy

Spec(\(\Pi, c\)) begin
\(\Pi_{Sp} = \emptyset\);
\(Def = \{c\}\);
while \(\exists q \in Def\) do
  \(Unf = \text{Clause Removal}( \text{Unfold}(q) )\);
  \(Def = Def - \{q\} \cup \text{Define}(Unf)\);
  \(\Pi_{Sp} = \Pi_{Sp} \cup \text{Fold}(Unf, Def)\)
done
end

Theorem: \(\Pi \models \varphi\) iff \(\Pi_{Sp} \models \varphi\)

- Generalizations in \(\text{Define}(\cdot)\) ensure termination of \(Spec\), but may prevent the proof of the property.
Software model checking
Framework Architecture

$P$ and $\varphi$ are encoded as $S$ and $prop$, respectively.
Specialization-based Software Model Checking
Verification Framework: SIMP language and safety properties

\[ a ::= n \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \times a_2 \]
\[ b ::= \text{true} \mid \text{false} \mid a_1 \ op \ a_2 \mid ! b \mid b_1 \ &\& \ b_2 \mid b_1 \mid\ b_2 \]
\[ t ::= ndc \mid b \]
\[ c ::= \text{skip} \mid x = a \mid c_1 ; c_2 \mid \text{if } t \text{ then } c_1 \text{ else } c_2 \mid \text{while } t \text{ do } c \text{ od} \]

CLP interpreter for the operational semantics of SIMP

\[
\begin{align*}
\text{tr}(s(\text{skip},S), E). \\
\text{tr}(s(\text{asgn}(\text{var}(X),A),E),s(\text{skip},E1)) &\iff \text{aeval}(A,S,V), \text{update}(\text{var}(X),V,S,E1). \\
\text{tr}(s(\text{comp}(C0,C1),S), s(C1,S1)) &\iff \text{tr}(s(C0,S),S1). \\
\text{tr}(s(\text{comp}(C0,C1),S),s(\text{comp}(C0',C1),S')) &\iff \text{tr}(s(C0,S),s(C0',S')). \\
\text{tr}(s(\text{ite}(B,C0,\_),S), s(C0,S)) &\iff \text{beval}(B,S). \\
\text{tr}(s(\text{ite}(B,\_,C1),S), s(C1,S)) &\iff \text{beval}(\text{not}(B),S). \\
\text{tr}(s(\text{ite}(ndc,S1,\_),E),s(S1,E)). \\
\text{tr}(s(\text{ite}(ndc,\_,S2),E),s(S3,E)). \\
\text{tr}(s(\text{while}(B,C),S), s(\text{ite}(B,\text{comp}(C,\text{while}(B,C)),\text{skip}),S)). \\
\end{align*}
\]
Specialization-based Software Model Checking

Verification Framework: SIMP language and safety properties

Let $P$ be a SIMP program and $\varphi$ be a safety property.

- **Phase 1:** Encode $P$ and $\varphi$ into a CLP program $\Pi$

\[
\begin{align*}
\text{reachable}(X) & :\neg \text{unsafe}(X). \\
\text{reachable}(X) & : t(X,X'), \text{reachable}(X'). \\
\text{unsafe} & : \text{initial}(X), \text{reachable}(X). \\
\text{unsafe}(s(\text{error},E)) & . \\
\text{initial}(s(T,E)) & : \text{init\_constraint}(E).
\end{align*}
\]

where:

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

- **Phase 2:** Spec - Specialize $\Pi$ w.r.t.

\[
\begin{align*}
\text{initial}(s(P,E)) & : \text{init\_constraint}(E).
\end{align*}
\]

- **Phase 3:** BuEval - Bottom up Evaluation of $\Pi_{Sp}$

$P$ is safe iff $\text{unsafe} \not\in \text{BuEval}(\Pi)$ iff $\text{unsafe} \not\in \text{BuEval}(\Pi_{Sp})$. 
Example

Phase 1: Encoding of $P$ and $\varphi$

```plaintext
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(
   ```plaintext
   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. \( \text{initial}(s(\text{comp}(\text{while}(\cdots),\cdots),[lv(x,X),\cdots])) : \ X=0,\cdots,N>0. \)
2. \( \text{unsafe}(s(\text{error},_)). \)

3. CLP Interpreter

\[
\begin{align*}
\text{new1}(X,Y,N) & : - X+1=\leq N, \ X'=X+1, \ Y'=Y+1, \ \text{new1}(X',Y',N). \\
\text{new1}(X,Y,N) & : - N=\leq X, \ Y> X. \\
\text{unsafe} & : - X=0, \ Y=0, \ N\geq 1, \ \text{new1}(X,Y,N). \\
\text{safe} & : - \not\text{unsafe}. 
\end{align*}
\]

\[
\begin{align*}
X+1=\leq N \\
\text{while} \quad N=\leq X, \ Y> X \\
\rightarrow \text{error}
\end{align*}
\]
Example

Phase 3: Bottom Up Evaluation of the Specialized Program

Let $\Pi_{Sp}$ the specialized CLP program:

\[
\text{new1}(X,Y,N) :- N \geq X+1, X'=X+1, Y'=Y+1, \text{new1}(X',Y',N). \\
\text{new1}(X,Y,N) :- N=X, Y'=X+1. \\
\text{unsafe} :- X=0, Y=0, N \geq 1, \text{new1}(X,Y,N). \\
\text{safe} :- \text{not unsafe.}
\]

$$\text{BuEval}(\Pi_{Sp}) = \{
\begin{align*}
\text{new1}(X,Y,N) :- & \quad X+1=\leq Y, \ N=\leq X. \\
\text{new1}(X,Y,N) :- & \quad X+1=\leq Y, \ N=X+1. \\
\text{new1}(X,Y,N) :- & \quad X+1=\leq Y, \ N=X+2. \\
\text{new1}(X,Y,N) :- & \quad X+1=\leq Y, \ N=X+3. \\
\text{new1}(X,Y,N) :- & \quad X+1=\leq Y, \ N=X+4. \\
\ldots & \}
\]$$

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 3: Bottom Up Evaluation of the Specialized Program

Let $\Pi_{Sp}$ the specialized CLP program:

\[
\begin{align*}
\text{new1}(X,Y,N) & :\ N\geq X+1, \ X'=X+1, \ Y'=Y+1, \ \text{new1}(X',Y',N). \\
\text{new1}(X,Y,N) & :\ N=X, \ Y'=X+1. \\
\text{unsafe} & :\ X=0, \ Y=0, \ N\geq 1, \ \text{new1}(X,Y,N). \\
\text{safe} & :\ \neg \text{unsafe}.
\end{align*}
\]

\[
\begin{align*}
\text{BuEval}(\Pi_{Sp}) & = \{ \\
\text{new1}(X,Y,N) & :\ X+1=\leq Y, \ N=\leq X. \\
\text{new1}(X,Y,N) & :\ X+1=\leq Y, \ N=X+1. \\
\text{new1}(X,Y,N) & :\ X+1=\leq Y, \ N=X+2. \\
\text{new1}(X,Y,N) & :\ X+1=\leq Y, \ N=X+3. \\
\text{new1}(X,Y,N) & :\ X+1=\leq Y, \ N=X+4. \\
\text{....} & \\
\}
\end{align*}
\]

The Bottom Up Evaluation does not terminate.
Example
Phase 3: Bottom Up Evaluation of the Specialized Program

Let \( \Pi_{Sp} \) the specialized CLP program:

\[
\begin{align*}
\text{new1}(X,Y,N) & : \quad N \geq X+1, \quad X' = X+1, \quad Y' = Y+1, \quad \text{new1}(X',Y',N). \\
\text{new1}(X,Y,N) & : \quad N = < X, \quad Y = X+1. \\
\text{unsafe} & : \quad X = 0, \quad Y = 0, \quad N \geq 1, \quad \text{new1}(X,Y,N). \\
\text{safe} & : \quad \neg \text{unsafe}. \\
\end{align*}
\]

\[
\text{BuEval}(\Pi_{Sp}) = \{ \\
\begin{align*}
\text{new1}(X,Y,N) & : \quad X+1 = < Y, \quad N = < X. \\
\text{new1}(X,Y,N) & : \quad X+1 = < Y, \quad N = X+1. \\
\text{new1}(X,Y,N) & : \quad X+1 = < Y, \quad N = X+2. \\
\text{new1}(X,Y,N) & : \quad X+1 = < Y, \quad N = X+3. \\
\text{new1}(X,Y,N) & : \quad X+1 = < Y, \quad N = X+4. \\
\end{align*}
\}
\]

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 $\Pi_{Sp}$

new1(X,Y,N) :- N $\geq$ 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 $\geq$1, new1(X,Y,N).
safe :- not unsafe.

\[\Downarrow\]

new2(X,Y,N) :- N $\geq$X, X’=X+1, Y’=Y+1, X’$\geq$Y’, Y’$\geq$1, new2(X’,Y’,N).
new1(X,Y,N) :- X=0, Y=0, N $\geq$1, Y’=1, X’=1, new2(X’,Y’,N).
unsafe :- X=0, Y=0, N $\geq$1, new1(X,Y,N).
safe :- not unsafe.
Example

Phase 2: Specialization of $\Pi_{Sp}$

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.

\[\text{new2}(X,Y,N) \text{ :- } N>=X, X'=X+1, Y'=Y+1, X'>=Y', Y'>=1, \text{new2}(X',Y',N).\]
\[\text{new1}(X,Y,N) \text{ :- } X=0, Y=0, N>=1, Y'=1, X'=1, \text{new2}(X',Y',N).\]
\[\text{unsafe} \text{ :- } X=0, Y=0, N>=1, \text{new1}(X,Y,N).\]
\[\text{safe} \text{ :- not unsafe.}\]

No facts
Example

Phase 2: Specialization of $\Pi_{Sp}$

\[
\text{new1}(X,Y,N) :- \quad N \geq X+1, \quad X' = X+1, \quad Y' = Y+1, \quad \text{new1}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- \quad N = < X, \quad X+1 = < Y.
\]
\[
\text{unsafe} :- \quad X = 0, \quad Y = 0, \quad N = > 1, \quad \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \quad \text{not unsafe}.
\]

\[
\text{new2}(X,Y,N) :- \quad N \geq X, \quad X' = X+1, \quad Y' = Y+1, \quad X' \geq Y', \quad Y' \geq 1, \quad \text{new2}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- \quad X = 0, \quad Y = 0, \quad N = > 1, \quad Y' = 1, \quad X' = 1, \quad \text{new2}(X',Y',N).
\]
\[
\text{unsafe} :- \quad X = 0, \quad Y = 0, \quad N = > 1, \quad \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \quad \text{not unsafe}.
\]

No facts

The Bottom Up Evaluation terminates
Example

Phase 2: Specialization of $\Pi_{Sp}$

\[
\text{new1}(X,Y,N) :- \ N \geq X+1, \ X' = X+1, \ Y' = Y+1, \ \text{new1}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- \ N = X, \ X+1 = Y.
\]
\[
\text{unsafe} :- \ X=0, \ Y=0, \ N \geq 1, \ \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \ \text{not unsafe}.
\]

\[
\downarrow
\]

\[
\text{new2}(X,Y,N) :- \ N \geq X, \ X' = X+1, \ Y' = Y+1, \ X' \geq Y', \ Y' \geq 1, \ \text{new2}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- \ X=0, \ Y=0, \ N \geq 1, \ Y' = 1, \ X' = 1, \ \text{new2}(X',Y',N).
\]
\[
\text{unsafe} :- \ X=0, \ Y=0, \ N \geq 1, \ \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \ \text{not unsafe}.
\]

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. ⊥ denotes ‘terminating with error’ (TRACER, using the default options, terminates with ‘Fatal Error: Heap overflow’). ∞ means ‘Model checking not successful within 20 minutes’.

<table>
<thead>
<tr>
<th>Programs</th>
<th>ARMC</th>
<th>TRACER</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1a</td>
<td>∞</td>
<td>⊥</td>
<td>0.08</td>
</tr>
<tr>
<td>f2</td>
<td>∞</td>
<td>⊥</td>
<td>7.58</td>
</tr>
<tr>
<td>Substring</td>
<td>719.39</td>
<td>180.09</td>
<td>10.20</td>
</tr>
<tr>
<td>prog_dagger</td>
<td>∞</td>
<td>⊥</td>
<td>5.37</td>
</tr>
<tr>
<td>seesaw</td>
<td>3.41</td>
<td>⊥</td>
<td>0.03</td>
</tr>
<tr>
<td>tracer_prog_d</td>
<td>∞</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>interpolants_needed</td>
<td>0.13</td>
<td>⊥</td>
<td>0.06</td>
</tr>
<tr>
<td>widen_needed</td>
<td>∞</td>
<td>⊥</td>
<td>0.07</td>
</tr>
</tbody>
</table>


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

- 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)