Software Model Checking by Program Specialization

Emanuele De Angelis\textsuperscript{1}, Fabio Fioravanti\textsuperscript{1}, Alberto Pettorossi\textsuperscript{2} and Maurizio Proietti\textsuperscript{3}

\textsuperscript{1}University of Chieti-Pescara ‘G. d’Annunzio’
\textsuperscript{2}University of Rome ‘Tor Vergata’
\textsuperscript{3}CNR-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 $\varphi$

\[ P \models \varphi \Downarrow \]

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 $\varphi$

$\alpha(P) \models \varphi \quad \Downarrow \quad P \models \varphi \quad \alpha$ refinement

$\alpha(P) \not\models \varphi$

Clarke et al. *CEGAR for Symbolic Model Checking.*
Cousot and Halbwachs. *Automatic Discovery of Linear Restraints Among Variables of a Program.*
Software model checking

Modelling software

abstraction $\alpha(P)$:

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

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

Etalle and Gabbrielli. Transforms of CLP modules.
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

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

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

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

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.
\hspace{1em} \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\hspace{1em} \( p(X_1, \ldots, X_n) \leftarrow c \land d \land A \)

R3 Folding  \( p(X_1, \ldots, X_n) \leftarrow c \land A \) w.r.t. \( A \) by using
\hspace{1em} \( q(X_1, \ldots, X_n) \leftarrow d \land A \)
gives
\hspace{1em} \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)  \hspace{1em} if \( c \Rightarrow d \)

R4 Clause removal

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

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

Software model checking

Specialization strategy

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

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

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

Source Code

FRONT-END

Lexer

Parser

S

BACK-END (MAP system)

Spec

Π

Π_{Sp}

Π_{Sp} \not\models prop

Spec

P \models prop

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

\[ a ::= n | x | a_1 + a_2 | a_1 - a_2 | a_1 \times a_2 \]

\[ b ::= \text{true} | \text{false} | a_1 \text{ op } a_2 | ! b | b_1 \text{ \&\& } b_2 | b_1 \text{ || } b_2 \]

\[ t ::= ndc | b \]

\[ c ::= \text{skip} | x = a | c_1; c_2 | \text{if } t \text{ then } c_1 \text{ else } c_2 | \text{while } t \text{ do } c \text{ od} \]

CLP interpreter for the operational semantics of SIMP

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

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 $\Pi$
  
  ```prolog
  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 $\Pi$ w.r.t.
  
  ```prolog
  initial(s(P,E)) :- init_constraint(E).
  ```

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

$$P \text{ is safe iff } unsafe \notin BuEval(\Pi) \iff unsafe \notin BuEval(\Pi_{Sp}).$$
Example
Phase 1: Encoding of \( P \) and \( \varphi \)

\[
\begin{align*}
\text{int } x &= 0; \quad \text{int } y = 0; \quad \text{int } n; \\
\text{assume}(n > 0); \\
\text{while } (x < n) \{ \, x = x + 1; \quad y = y + 1; \, \} \\
\text{if } (y > x) \text{ error;}
\end{align*}
\]

\[
\begin{align*}
1. \quad \text{initial(} \\
\quad &\text{\quad s(comp(while(lt(var(x),var(n)),} \\
\quad &\quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad comp(asgn(var(x),plus(var(x),int(1)))))}, \\
\quad &\quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad asgn(var(y),plus(var(y),int(1))))),} \\
\quad &\quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad ite(gt(var(y),var(x)),error,skip)),} \\
\quad &\quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad [lv(x,X),lv(y,Y),lv(n,N)]) :} \\
\quad &\quad \text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad X=0,Y=0,N>0.} \\
2. \quad \text{unsafe(s(error,_)).}
\end{align*}
\]
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.

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

Phase 3: Bottom Up Evaluation of the Specialized Program

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

\[
\text{new1}(X,Y,N) :- \ N>=X+1, \ X'=X+1, \ Y'=Y+1, \ \text{new1}(X',Y',N).
\]

\[
\text{new1}(X,Y,N) :- \ N=<X, \ Y'=X+1.
\]

\[
\text{unsafe} :- \ X=0, \ Y=0, \ N>=1, \ \text{new1}(X,Y,N).
\]

\[
\text{safe} :- \ \neg \text{unsafe}.
\]

\[
\text{BuEval}(\Pi_{Sp}) = \{
\begin{array}{l}
\text{new1}(X,Y,N) :- \ X+1=<Y, \ N=<X.
\text{new1}(X,Y,N) :- \ X+1=<Y, \ N=X+1.
\text{new1}(X,Y,N) :- \ X+1=<Y, \ N=X+2.
\text{new1}(X,Y,N) :- \ X+1=<Y, \ N=X+3.
\text{new1}(X,Y,N) :- \ X+1=<Y, \ N=X+4.
\end{array}
\]

\[
\text{.... } \}
\]
Example

Phase 3: Bottom Up Evaluation of the Specialized Program

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

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

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

The Bottom Up Evaluation does not terminate.
Example

Phase 3: Bottom Up Evaluation of the Specialized Program

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

\[
\text{new1}(X,Y,N) :- \ N \geq X+1, \ X' = X+1, \ Y' = Y+1, \ \text{new1}(X',Y',N).
\]

\[
\text{new1}(X,Y,N) :- \ N \leq X, \ Y \geq X+1.
\]

\[
\text{unsafe} :- \ X = 0, \ Y = 0, \ N \geq 1, \ \text{new1}(X,Y,N).
\]

\[
\text{safe} :- \ \text{not unsafe}.
\]

\[
\text{BuEval}(\Pi_{Sp}) = \{
\]

\[
\text{new1}(X,Y,N) :- \ X+1 = \leq Y, \ N = \leq X.
\]

\[
\text{new1}(X,Y,N) :- \ X+1 = \leq Y, \ N = X+1.
\]

\[
\text{new1}(X,Y,N) :- \ X+1 = \leq Y, \ N = X+2.
\]

\[
\text{new1}(X,Y,N) :- \ X+1 = \leq Y, \ N = X+3.
\]

\[
\text{new1}(X,Y,N) :- \ X+1 = \leq Y, \ N = X+4.
\]

\[
\ldots \}
\]

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 2: Specialization of $\Pi_{Sp}$

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

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

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

\[
\text{new1}(X,Y,N) :- N \geq X+1, X'=X+1, Y'=Y+1, \text{new1}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- N<X, X+1=Y.<Y.
\]
\[
\text{unsafe} :- X=0, Y=0, N\geq1, \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \text{not unsafe}.
\]

\[
\downarrow
\]

\[
\text{new2}(X,Y,N) :- N \geq X, X'=X+1, Y'=Y+1, X'=Y', Y'=1, \text{new2}(X',Y',N).
\]
\[
\text{new1}(X,Y,N) :- X=0, Y=0, N=1, Y'=1, X'=1, \text{new2}(X',Y',N).
\]
\[
\text{unsafe} :- X=0, Y=0, N\geq1, \text{new1}(X,Y,N).
\]
\[
\text{safe} :- \text{not unsafe}.
\]

No facts
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.

\[
\downarrow
\]

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 $\Pi_{Sp}$

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

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

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’).
\(\infty\) 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>(\infty)</td>
<td>(\perp)</td>
<td>0.08</td>
</tr>
<tr>
<td>f2</td>
<td>(\infty)</td>
<td>(\perp)</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>(\infty)</td>
<td>(\perp)</td>
<td>5.37</td>
</tr>
<tr>
<td>seesaw</td>
<td>3.41</td>
<td>(\perp)</td>
<td>0.03</td>
</tr>
<tr>
<td>tracer_prog_d</td>
<td>(\infty)</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>interpolants_needed</td>
<td>0.13</td>
<td>(\perp)</td>
<td>0.06</td>
</tr>
<tr>
<td>widen_needed</td>
<td>(\infty)</td>
<td>(\perp)</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

  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)