Verifying Programs via Iterated 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 - Istituto di Analisi dei Sistemi ed Informatica, Rome

PEPM 2013
Rome, Italy, 21 January 2013
Software Model Checking of imperative programs...

  - Safety properties of C programs

... by iterated specialization of Constraint Logic Programs

  - First specialization: removal of the interpreter
  - Subsequent specializations:
    one or more propagations of constraints of the initial or error configurations

Experimental results
A program $P$ is safe w.r.t. $\varphi_{init}$ and $\varphi_{error}$ if from any configuration satisfying $\varphi_{init}$ no configuration satisfying $\varphi_{error}$ can be reached. Otherwise, program $P$ is unsafe.
Program execution as a transition relation.

Program $P$:

```c
void main(){
    int x;
    int y;
    int n;

    $\ell_0$: while(x < n) {
        $\ell_1$: x = x+1;
        $\ell_2$: y = x+y;
    }

    $\ell_h$: }
```

Execution of $P$:

- Initial state: $x = 0 \land y = 0$
- Transition from $\ell_0$ to $\ell_1$: $x \geq n$
- Transition from $\ell_0$ to $\ell_2$: $x < n$
- Transition from $\ell_1$: $y = x+y$
- Transition from $\ell_2$: $x = x+1$

- (unsafe config.) $x > y$
- (safe config.) $x \leq y$
Related Work

- **Static analysis and model checking**
  - Saïdi. *Model checking guided abstraction and analysis.* [SAS’00]

- **Constraint-based verification**
  - Podolski and Rybalchenko. *ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement.* [PADL’07]
  - Grebenshchikov, Gupta, Lopes, Popeea, and Rybalchenko. *HSF(C): A Software Verifier based on Horn Clauses.* [TACAS’12]

- **Specialization-based verification**
  - Peralta, Gallagher, and Saglam. *Analysis of Imperative Programs through Analysis of Constraint Logic Programs.* [SAS’98]
Verification Framework

We use a **Constraint Logic Program** (CLP) program for encoding:

- the program $P$ to be verified (written in the language C)
- the interpreter $\text{Int}$ (i.e., the semantics of the language C)
- the configurations $\varphi_{\text{init}}$ or $\varphi_{\text{error}}$

\[ \text{Int} \]

\[ \text{Specialize w.r.t. } P \text{ (removal of the interpreter)} \]

\[ \text{SpInt} \]

\[ \text{Analyze} \]

\[ \text{Specialize w.r.t. } \varphi_{\text{init}} \text{ or } \varphi_{\text{error}} \text{ one or more times} \]

- $P$ unsafe + counterexample
- $P$ safe

\[ (\text{In general, safety is undecidable}) \]
program P:

```c
void main()
{
    int x;
    int y;
    int n;

    ℓ₀: while(x < n) {
        ℓ₁:   x=x+1;
        ℓ₂:   y=x+y;
        ℓ₃: }
    ℓ₄: }
```

encoding of program P:

```c
at(ℓ₀, ite(less(int(x), int(n)), ℓ₁, ℓ₄)).
at(ℓ₁, asgn(int(x), plus(int(x), int(1))))).
at(ℓ₂, asgn(int(y), plus(int(x), int(y))))).
at(ℓ₃, goto(ℓ₀)).
at(ℓ₄, halt).
```
a set of configurations: \( \text{cf}(C, S) \)

Each configuration is made out of:
- a **command** \( C \)
- a **state** \( S \): a list of [variable, value] pairs
  
  for instance: \([\text{int}(x), x_1], [\text{int}(y), y_1]\]

a transition relation: \( \text{tr}(\text{cf}(C, S), \text{cf}(C_1, S_1)) \) \( \rightarrow \)

(i.e., operational semantics)
<table>
<thead>
<tr>
<th>Expression</th>
<th>Prolog Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Id = Expr;</code></td>
<td><code>tr(cf(L, asgn(Id, Expr), S), cf(C, S1)) :- aeval(Expr, S, V), update(Id, V, S, S1), nextlab(L, C).</code></td>
</tr>
<tr>
<td><code>if (Expr) {</code></td>
<td><code>tr(cf(ite(Expr, L1, L2), S), cf(C, S)) :- beval(Expr, S), at(L1, C).</code></td>
</tr>
<tr>
<td><code>goto L1;</code></td>
<td><code>tr(cf(ite(Expr, L1, L2), S), cf(C, S)) :- beval(not(Expr), S), at(L2, C).</code></td>
</tr>
<tr>
<td><code>} else</code></td>
<td><code>goto L;</code></td>
</tr>
<tr>
<td><code>goto L2;</code></td>
<td><code>tr(cf(goto(L), S), cf(C, S)) :- at(L, C).</code></td>
</tr>
<tr>
<td><code>}</code></td>
<td><code>Id = F(ArgList);</code></td>
</tr>
<tr>
<td><code>tr(cf(call(F, ArgList, Id, Ret), S), cf(goto(Ep), S1)) :- prologue(F, ArgList, S, Id, Ret, Ep, S1).</code></td>
<td></td>
</tr>
<tr>
<td><code>return Expr;</code></td>
<td><code>tr(cf(ret(Expr), S), cf(C, S1)) :- epilogue(Expr, S, S1, Ret), at(Ret, C).</code></td>
</tr>
</tbody>
</table>
Theorem: Program P is safe iff the atom unsafe does not belong to the least model $M(\text{Int})$ of the CLP program Int.
Program specialization is a program manipulation technique whose objective is the adaptation of a program to a context of use.

It is based on transformation rules.

It allows an agile development of verification tools because:

- it is parametric w.r.t. languages and logics
- it allows the composition of various program transformations
- transformation rules for specialization:
  \( R \in \{ \text{Definition Introduction, Unfolding, Folding, Clause Removal} \} \)

- rules are semantic preserving:
  \[ \text{unsafe} \in M(\text{Int}) \iff \text{unsafe} \in M(\text{SplInt}) \]

- specialization strategy:
  \((\text{Unfolding}; \text{Clause Remov}; \text{Def Intro}; \text{Folding})^*\)
Rules for Specializing CLP Programs

R1. Definition Introduction: \[ \text{newp}(X_1, \ldots, X_n) \leftarrow c \land A \]

R2. Unfolding: \[ p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \]
\[ q(X_1, \ldots, X_n) \leftarrow d_1 \land A_1, \ldots, q(X_1, \ldots, X_n) \leftarrow d_m \land A_m \]
yields \[ p(X_1, \ldots, X_n) \leftarrow c \land d_1 \land A_1, \ldots, p(X_1, \ldots, X_n) \leftarrow c \land d_m \land A_m \]

R3. Folding: \[ p(X_1, \ldots, X_n) \leftarrow c \land A \]
\[ q(X_1, \ldots, X_n) \leftarrow d \land A \quad \text{and} \quad c \rightarrow d \]
yields \[ p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \]

R4. Clause Removal:
if \( c \) is unsatisfiable or \( (p(X_1, \ldots, X_n) \leftarrow d \land c \rightarrow d) \),
then remove \( p(X_1, \ldots, X_n) \leftarrow c \land q(X_1, \ldots, X_n) \)
Let `unsafe` be a clause of the form: `unsafe:-Body`.

Specialize(`Int, unsafe`) 

\[
\begin{align*}
\text{SplInt} &= \emptyset; \\
\text{Def} &= \{\text{unsafe}\}; \\
\text{while } \exists q \in \text{Def} \text{ do} \\
\text{Unf} &= \text{Clause Removal}(\text{Unfold}(q)); \\
\text{Def} &= (\text{Def} - \{q\}) \cup \text{Define}(\text{Unf}); \\
\text{SplInt} &= \text{SplInt} \cup \text{Fold}(\text{Unf}, \text{Def}); \\
\end{align*}
\]

- `P` is safe iff `unsafe \notin M(\text{SplInt})`,
- Define realizes different specializations (w.r.t. `P`, `\varphi_{\text{init}}`, and `\varphi_{\text{error}}`),
- generalizations in Define ensure termination.
Compile away the C interpreter, i.e., remove all references to:

- **tr** (i.e., the operational semantics of C)
- **at** (i.e., the encoding of P)

Specialize **Int** w.r.t. **P**

\[
\begin{align*}
\text{SpInt:} & \quad \text{unsafe} : \ - X = 0, \ Y = 0, \ \text{new1}(X, Y, N). \\
& \text{new1}(X,Y,N) : \ - X < N, \ X1 = X + 1, \ Y1 = X1 + Y, \ \text{new1}(X1,Y1,N). \\
& \text{new1}(X,Y,N) : \ - X \geq N, \ X > Y.
\end{align*}
\]
Analyzing $\text{SpInt}$ to check safety of $P$:

- $P$ is safe iff $\text{unsafe} \not\in M(\text{SpInt})$,
- checking whether or not $\text{unsafe}$ belongs to $M(\text{SpInt})$ is undecidable,

looking for constrained facts:

- no constrained facts implies $M(\text{SpInt}) = \emptyset$,
- $M(\text{SpInt}) = \emptyset$ implies that $P$ is safe,
- very efficient,
- precision achieved by iterated specialization.
Analyze Splnt

We only look for constrained facts in Splnt:

\[
\text{unsafe} : - X=0,\ Y=0,\ \text{new1}(X, Y, N).
\]
\[
\text{new1}(X, Y, N) : - X < N,\ X_1 = X + 1,\ Y_1 = X_1 + Y,\ \text{new1}(X_1, Y_1, N).
\]
\[
\text{new1}(X, Y, N) : - X \geq N,\ X > Y.
\]

Splnt has a constrained fact for new1

At this point we cannot show that unsafe does not hold.

\[
\text{Splnt}
\]

\[
\begin{array}{c}
\text{Analyze} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Specialize w.r.t.} \\
\end{array}
\]

\[
\begin{array}{c}
\varphi_{\text{init}} \text{ or } \varphi_{\text{error}} \\
\end{array}
\]

\[
\begin{array}{c}
\text{un-safe} \\
\end{array}
\]

\[
\begin{array}{c}
\text{safe} \\
\end{array}
\]

\[
\begin{array}{c}
\text{counterexample} \\
\end{array}
\]

We need further specializations.
The output of Specialize, i.e., SpInt

\[ \text{unsafe} :- X = 0, \ Y = 0, \ \text{new1}(X, Y, N). \]
\[ \text{new1}(X, Y, N) :- X < N, \ X1 = X + 1, \ Y1 = X1 + Y, \ \text{new1}(X1, Y1, N). \]
\[ \text{new1}(X, Y, N) :- X \geq N, \ X > Y. \]

can be viewed as a transition system:

\[ \text{initial}((\text{new1}, X, Y, N)) :- X = 0, \ Y = 0. \]
\[ \text{tr}((\text{new1}, X, Y, N), (\text{new1}, X1, Y1, N)) :- X < N, \ X1 = X + 1, \ Y1 = X1 + Y. \]
\[ \text{error}((\text{new1}, X, Y, N)) :- X \geq N, \ X > Y. \]

By specializing:

\[ \text{unsafe} :- \text{initial}(A), \ \text{reach}(A). \]
\[ \text{reach}(A) :- \text{tr}(A, B), \ \text{reach}(B). \]
\[ \text{reach}(X) :- \text{error}(A). \]

w.r.t. \text{unsafe}, we propagate the constraint \( X = 0, \ Y = 0 \) of the initial configuration \( \varphi_{init} \).
Propagation of the initial configuration

Propagation of the constraint $X=0$, $Y=0$.

$\text{SpInt} \quad \text{new} \quad \text{SpInt}$

Specialize w.r.t. $\phi_{\text{init}}$

$(\text{new}) \text{SpInt}$
The output of Specialize, i.e., $\text{SplInt}$

\begin{align*}
\text{unsafe} &: - N > 0, \ X_1 = 1, \ Y_1 = 1, \ \text{new2}(X_1, Y_1, N) . \\
\text{new2}(X, Y, N) &: - X = 1, \ Y = 1, \ N > 1, \ X_1 = 2, \ Y_1 = 3, \ \text{new3}(X_1, Y_1, N) . \\
\text{new3}(X, Y, N) &: - X_1 \geq 1, \ Y_1 \geq X_1, \ X < N, \ X_1 = X + 1, \ Y_1 = X_1 + Y, \ \text{new3}(X_1, Y_1, N) . \\
\text{new3}(X, Y, N) &: - Y \geq 1, \ N > 0, \ X \geq N, \ X > Y .
\end{align*}

can be viewed as a transition system:

\begin{align*}
\text{initial}((\text{new1}, X, Y, N)) &: - N > 0, \ X_1 = 1, \ Y_1 = 1 . \\
\text{tr}((\text{new2}, X, Y, N), (\text{new3}, X_1, Y_1, N)) &: - X = 1, \ Y = 1, \ N > 1, \ X_1 = 2, \ Y_1 = 3 . \\
\text{tr}((\text{new3}, X, Y, N), (\text{new3}, X_1, Y_1, N)) &: - X_1 \geq 1, \ Y_1 \geq X_1, \ X < N, \ X_1 = X + 1, \ Y_1 = X_1 + Y . \\
\text{error}((\text{new3}, X, Y, N)) &: - Y \geq 1, \ N > 0, \ X \geq N, \ X > Y .
\end{align*}

In order to propagate the constraint of the error configuration $\varphi_{\text{error}}$ we reverse the direction of the reachability relation $\text{reach}$. 
Program Reversal

By specializing

\[
\begin{align*}
\text{unsafe} & :\ = \ initial(A), \ reach(A). \\
\text{reach}(A) & :\ = \ tr(A,B), \ reach(B). \\
\text{reach}(X) & :\ = \ error(A).
\end{align*}
\]

w.r.t. \texttt{unsafe}, we propagate the constraint of the initial configuration \(\varphi_{init}\).

By specializing

\[
\begin{align*}
\text{unsafe} & :\ = \ error(A), \ reach(A). \\
\text{reach}(B) & :\ = \ tr(A,B), \ reach(A). \\
\text{reach}(X) & :\ = \ initial(A).
\end{align*}
\]

w.r.t. \texttt{unsafe}, we propagate the constraint of the error configuration \(\varphi_{error}\).

\(\text{unsafe} \in M(\text{SplInt}) \iff \text{unsafe} \in M(\text{RevSplnt})\)
Propagation of the constraint $X > Y$.

\[
\text{unsafe} : - \ N > 0, \ X_1 = 1, \ Y_1 = 1, \ \text{new2}(X_1, Y_1, N).
\]
\[
\text{new2}(X, Y, N) : - X = 1, \ Y = 1, \ N > 1, \ X_1 = 2, \ Y_1 = 3, \ \text{new3}(X_1, Y_1, N).
\]
\[
\text{new3}(X, Y, N) : - X_1 \geq 1, \ Y_1 \geq X_1, \ X < N, \ X_1 = X + 1, \ Y_1 = X_1 + Y, \ \text{new3}(X_1, Y_1, N).
\]
\[
\text{new3}(X, Y, N) : - Y \geq 1, \ N > 0, \ X \geq N, \ X > Y.
\]

\[
\text{unsafe} : - Y \geq 1, \ N > 0, \ X \geq N, \ X > Y, \ \text{new4}(X, Y, N).
\]
Fully automatic Software Model Checker for proving safety of C programs.

- **CIL (C Intermediate Language)**

- **MAP Transformation System**
  [http://map.uniroma2.it/mapweb](http://map.uniroma2.it/mapweb)
## Experimental result

http://map.uniroma2.it/smc/

<table>
<thead>
<tr>
<th>Program</th>
<th>MAP(a)</th>
<th>MAP(b)</th>
<th>ARMC</th>
<th>HSF(C)</th>
<th>TRACER</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>n</td>
<td></td>
<td></td>
<td>SPost</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>WPre</td>
</tr>
<tr>
<td>barber1</td>
<td>1</td>
<td>13.71</td>
<td>2</td>
<td>26.43</td>
<td>414.01</td>
</tr>
<tr>
<td>berkeley</td>
<td>1</td>
<td>1.57</td>
<td>2</td>
<td>1.53</td>
<td>11.28</td>
</tr>
<tr>
<td>efm</td>
<td>3</td>
<td>6.48</td>
<td>2</td>
<td>4.04</td>
<td>31.17</td>
</tr>
<tr>
<td>ex1</td>
<td>1</td>
<td>0.03</td>
<td>2</td>
<td>0.40</td>
<td>1.69</td>
</tr>
<tr>
<td>f1a</td>
<td>2</td>
<td>0.17</td>
<td>1</td>
<td>0.07</td>
<td>-</td>
</tr>
<tr>
<td>heapSort</td>
<td>1</td>
<td>8.16</td>
<td>2</td>
<td>13.51</td>
<td>39.66</td>
</tr>
<tr>
<td>heapSort1</td>
<td>1</td>
<td>3.01</td>
<td>2</td>
<td>9.58</td>
<td>20.55</td>
</tr>
<tr>
<td>interp</td>
<td>1</td>
<td>0.12</td>
<td>2</td>
<td>0.28</td>
<td>11.41</td>
</tr>
<tr>
<td>lifo</td>
<td>1</td>
<td>20.56</td>
<td>2</td>
<td>15.59</td>
<td>126.54</td>
</tr>
<tr>
<td>p2</td>
<td>1</td>
<td>14.75</td>
<td>1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>rel</td>
<td>1</td>
<td>0.23</td>
<td>1</td>
<td>0.08</td>
<td>-</td>
</tr>
<tr>
<td>selectSort</td>
<td>3</td>
<td>1.96</td>
<td>6</td>
<td>3.26</td>
<td>24.97</td>
</tr>
<tr>
<td>singleLoop</td>
<td>3</td>
<td>0.35</td>
<td>2</td>
<td>0.28</td>
<td>-</td>
</tr>
<tr>
<td>substring</td>
<td>2</td>
<td>0.16</td>
<td>1</td>
<td>0.20</td>
<td>472.32</td>
</tr>
<tr>
<td>tracerP</td>
<td>1</td>
<td>0.01</td>
<td>1</td>
<td>0.07</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>#verified programs</td>
<td>20 (9)</td>
<td>19 (15)</td>
<td>13</td>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>total time</td>
<td>353.29</td>
<td>209.94</td>
<td>1971.10</td>
<td>69.42</td>
<td>23.33</td>
</tr>
</tbody>
</table>

Time (in seconds). ‘-’ means ‘unable to verify within 10 min’. $n$ is the number of specializations performed by the MAP system (after removal of the interpreter).
Software Model Checking framework, which is parametric w.r.t.:
- the language of the programs to be verified,
- the logic of the property to be checked.

Future Work:
- more features of the C language (arrays, pointers, etc.),
- more complex properties (e.g., liveness properties)
- different languages (e.g., Java and C#)
  to deal with object-oriented features and concurrency.