Software Verification and Synthesis using Constraints and Program Transformation

Emanuele De Angelis

University ‘G. d’Annunzio’ of Chieti–Pescara

Convegno Italiano di Logica Computazionale 2015
Genova, 1 July, 2015
Summary

Verification framework
- Sequential Programs (e.g., C programs)
- Formal language: Constraint Logic Programming
- Proof technique: Program Transformation

Implementation and Experimental Results
- the VeriMAP tool

Synthesis framework
- Concurrent Programs (e.g., Peterson algorithm)
- Formal language: Answer Set Programming
- Synthesis technique: Answer Set Solvers
Verification Conditions as CLP Programs

Given the program \textit{prog} and the specification \( \{ \varphi_{init} \} \text{prog} \{ \neg \varphi_{error} \} \)

\[
\{ x=0 \land y=0 \land n \geq 1 \} \\
\text{while}(x < n) \{ \\
\quad x = x + 1; \\
\quad y = y + 2; \\
\} \\
\{ y > x \}
\]

Verification Conditions (VCs) can be encoded as a set of clauses \( P \)

\begin{align*}
\text{incorrect} & :- X = 0, \ Y = 0, \ N \geq 1, \ \text{while}(X, \ Y, \ N). \quad \text{Initialization} \\
\text{while}(X, \ Y, \ N) & :- X < N, \ X_1 = X + 1, \ Y_1 = Y + 2, \ \text{while}(X_1, Y_1, N). \quad \text{Loop body} \\
\text{while}(X, \ Y, \ N) & :- X \geq N, \ Y \leq X. \quad \text{Exit}
\end{align*}

VCs are satisfiable iff \textit{incorrect} not in the least model \( M(P) \) of \( P \)

How to (automatically)
(A) generate the VCs for \textit{prog} ?
(B) prove the satisfiability of the VCs ?
The Transformation-based Verification Method

Transformation of **Constraint Logic Programs** (CLP) to:
- **generate** the Verification Conditions (VCs)
- **prove** the satisfiability of the VCs

**Interpreter:** \( \text{Int} \)

**Specification:** \( \{ \varphi_{\text{init}} \} \ prog \ \{ \neg \varphi_{\text{error}} \} \)

**Verification Conditions:** VCs

1. **Encode into CLP**
2. **Specialize \( \text{Int} \) w.r.t. \( prog \)** (generate the VCs)
3. **Propagate \( \varphi_{\text{init}} \) or \( \varphi_{\text{error}} \) and Analyze** (prove the satisfiability of the VCs)

**Verification method:** \( (1); (2); (3)^+ \)
Proof rules for safety (reachability of error configurations)

\[
\text{incorrect} :- \text{initial}(X), \text{phiInit}(X), \text{reach}(X).
\]
\[
\text{reach}(X) :- \text{tr}(X,Y), \text{reach}(Y).
\]
\[
\text{reach}(X) :- \text{final}(X), \text{phiError}(X).
\]

Operational semantics of the programming language

\[
\text{tr}(\text{cf}(\text{Lab1},\text{Cmd1}),\text{cf}(\text{Lab2},\text{Cmd2})) :- \ldots
\]

E.g., operational semantics of conditionals

\[
\begin{align*}
\text{L: if(Expr) \{} & \quad \text{tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-} \\
& \quad \quad \text{beval(Expr,S),} \quad \quad \% \text{expression is true} \\
& \quad \quad \text{at(L1,C).} \quad \quad \% \text{next command} \\
& \quad \text{else} \\
& \quad \quad \text{tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-} \\
& \quad \quad \quad \text{beval(not(Expr),S),} \quad \quad \% \text{expression is false} \\
& \quad \quad \quad \text{at(L2,C).} \quad \quad \% \text{next command}
\end{align*}
\]

Theorem (Correctness of Encoding)

\[
\text{prog is correct \iff incorrect \notin M(\text{Int}) (the least model of Int)}
\]
Given the program \textit{prog} and the specification \{\(\phi_{\text{init}}\) \textit{prog} \{\(\neg\phi_{\text{error}}\}\}

\[
\{ x=0 \land y=0 \land n \geq 1 \}
\]

\[
\text{while}(x < n) \{ \\
\quad x = x + 1; \\
\quad y = y + 2; \\
\}
\]

\[
\{ y > x \}
\]

CLP encoding of \(\phi_{\text{init}}\) and \(\phi_{\text{error}}\)

\begin{align*}
\text{phiInit}(X, Y, N) &: \neg X = 0, Y = 0, N \geq 1. \\
\text{phiError}(X, Y) &: Y \leq X.
\end{align*}

CLP encoding of program \textit{prog}

A set of \texttt{at}(label, command) facts. while commands are replaced by \texttt{ite} and \texttt{goto}.

\begin{align*}
\text{at}(0, \texttt{ite}(< x, n, 1, h)). \\
\text{at}(1, \texttt{asgn}(x, +x, 1)). \\
\text{at}(2, \texttt{asgn}(y, +y, 2)). \\
\text{at}(3, \texttt{goto}(0)). \\
\text{at}(h, \texttt{halt}).
\end{align*}
The specialization of $\text{Int}$ w.r.t. $\text{prog}$ removes all references to:

- $\text{tr}$ (i.e., the operational semantics of the imperative language)
- $\text{at}$ (i.e., the encoding of $\text{prog}$)

The Specialized Interpreter for $\text{prog}$ (Verification Conditions)

\[
\text{incorrect} :- X = 0, \ Y = 0, \ N \geq 1, \ \text{while}(X, Y, N).
\]
\[
\text{while}(X, Y, N) :- X < N, \ X_1 = X + 1, \ Y_1 = Y + 2, \ \text{while}(X_1, Y_1, N).
\]
\[
\text{while}(X, Y, N) :- X \geq N, \ Y \leq X.
\]

New predicates correspond to a subset of the program points:

\[
\text{while}(X, Y, N) :- \text{reach}(\text{cf}(\text{cmd}(0, \text{ite}(\text{...}))),
\quad [[\text{int}(x), X], [[\text{int}(y), Y], [[\text{int}(n), N]]]).
\]
Rule-based program transformation

- transformation rules:
  \[ R \in \{ \text{Unfolding, Clause Removal, Definition, Folding} \} \]

- the transformation rules preserve the least model:

\[ \text{incorrect} \in M(P) \iff \text{incorrect} \in M(\text{TransfP}) \]

- the rules must be guided by a strategy.
The Unfold/Fold Transformation Strategy

Transform\((P)\)

\[
TransfP = \emptyset;
\]
\[
\text{Defs} = \{\text{incorrect} :- \text{initial}(X), \text{phiInit}(X), \text{reach}(X)\};
\]
\[
\text{while } \exists q \in \text{Defs} \text{ do}
\]
\[
\text{% execute a symbolic evaluation step (resolution)}
\]
\[
\text{Cls} = \text{Unfold}(q);
\]
\[
\text{% remove unsatisfiable and subsumed clauses}
\]
\[
\text{Cls} = \text{ClauseRemoval}(\text{Cls});
\]
\[
\text{% introduce new predicates (e.g., a loop invariant)}
\]
\[
\text{Defs} = (\text{Defs} - \{q\}) \cup \text{Define}(\text{Cls});
\]
\[
\text{% match a predicate definition}
\]
\[
TransfP = TransfP \cup \text{Fold}(\text{Cls}, \text{Defs});
\]
\[
\text{od}
\]
Propagation of $\varphi_{init}$

The transformation of the VCs $P$

VCs for $\text{prog}$ (Specialized interpreter $\text{Int}$)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>incorrect :- X=0, Y=0, N ≥ 1, while(X,Y,N).</td>
<td></td>
</tr>
<tr>
<td>while(X,Y,N):- X &lt; N, X1=X+1, Y1=Y+2, while(X1,Y1,N).</td>
<td></td>
</tr>
<tr>
<td>while(X,Y,N) :- X ≥ N, Y ≤ X.</td>
<td></td>
</tr>
</tbody>
</table>

by propagating the constraint $X = 0, Y = 0, N \geq 1$, modifies the structure of $P$ and derives the new VCs $\text{TransfP}$

Transformed VCs for $\text{prog}$

<table>
<thead>
<tr>
<th>Rule</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>incorrect :- X=0, Y=0, N ≥ 1 new1(X,Y,N).</td>
<td></td>
</tr>
<tr>
<td>new1(X,Y,N) :- X=0, Y=0, N ≥ 1, X1=1, Y1=2, new2(X1,Y1,N).</td>
<td></td>
</tr>
<tr>
<td>new2(X,Y,N) :- X &lt; N, X1=X+1, Y1=Y+2, X1 ≥ 1, Y1 ≥ 2, new2(X1,Y1,N).</td>
<td></td>
</tr>
<tr>
<td>new2(X,Y,N) :- X ≥ N, Y ≤ X, Y ≥ 0, N ≥ 1.</td>
<td></td>
</tr>
</tbody>
</table>

The fact incorrect is not in $\text{TransfP}$, we cannot infer that $\text{prog}$ is incorrect. A constrained fact is in $\text{TransfP}$, we cannot infer that $\text{prog}$ is correct.
Propagation of $\varphi_{error}$

Transformed VCs for prog (after the propagation of $\varphi_{init}$)

- incorrect :- $X = 0$, $Y = 0$, $N \geq 1$, new1($X$, $Y$, $N$).
- new1($X$, $Y$, $N$) :- $X = 0$, $Y = 0$, $N > 1$, $X1 = 1$, $Y1 = 2$, new2($X1$, $Y1$, $N$).
- new2($X$, $Y$, $N$) :- $X < N$, $X1 = X + 1$, $Y1 = Y + 2$, $X1 \geq 1$, $Y1 \geq 2$, new2($X1$, $Y1$, $N$).
- new2($X$, $Y$, $N$) :- $X \geq N$, $Y \leq X$, $Y \geq 0$, $N \geq 1$.

Reversed VCs

- incorrect :- $X \geq N$, $Y \leq X$, $Y \geq 0$, $N \geq 1$, new2($X$, $Y$, $N$).
- new2($X1$, $Y1$, $N$) :- $X = 0$, $Y = 0$, $N > 1$, $X1 = 1$, $Y1 = 2$, new1($X$, $Y$, $N$).
- new2($X1$, $Y1$, $N$) :- $X < N$, $X1 = X + 1$, $Y1 = Y + 2$, $X1 \geq 1$, $Y1 \geq 2$, new2($X$, $Y$, $N$).
- new1($X$, $Y$, $N$) :- $X = 0$, $Y = 0$, $N \geq 1$.

by propagating $\varphi_{error}$, that is, the constraint $X \geq N$, $Y \leq X$, $Y \geq 0$, $N \geq 1$.

Transformed VCs for prog (after the propagation of $\varphi_{error}$)

- incorrect :- $X \geq N$, $Y \leq X$, $Y \geq 0$, $N \geq 1$, new3($X$, $Y$, $N$).
- new3($X1$, $Y1$, $N$) :- $X < N$, $X1 = X + 1$, $Y1 = Y + 2$, $X > Y$, $Y \geq 0$, new3($X$, $Y$, $N$).

No facts: prog is correct.
Verification Framework

Step (0) **Translate** \( \text{Prog} \) and \( \varphi \) into CLP

\[ \text{Source to CLP Translator} \]

Program \( \text{prog} \)
(written in \( L \))

Specfication \( \varphi \)
(specified in \( M \))

Initial CLP Program \( T \)

Interpreter \( \text{Int} \)
(Semantics of \( L \))
(Semantics of \( M \))

Step (1) **Specialize** \( \text{Int} \) w.r.t. \( T \)
(Removal of the Interpreter)

**Verification Condition Generator**

Verification Conditions (VC’s) \( V \)

Step (2) **Transform** verification conditions w.r.t. \( \varphi \)

**Unfold/Fold Transformer**

Transformed VC’s \( S \)

Step (3) **Check** whether or not \( \varphi \) holds in \( Q \)

**Analyzer**

true
false

unknown

+ \( S \)
Fully automatic software model checker for C programs.

- **CIL** (C Intermediate Language) by Necula et al.
- **MAP Transformation System** by the MAP group (IASI-CNR, ‘G. d’Annunzio’ and ‘Tor Vergata’ Universities)
216 examples taken from: DAGGER, TRACER, InvGen, and TACAS 2013 Software Verification Competition.

<table>
<thead>
<tr>
<th></th>
<th>VeriMAP</th>
<th>ARMC</th>
<th>HSF(C)</th>
<th>TRACER</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 correct answers</strong></td>
<td>185</td>
<td>138</td>
<td>160</td>
<td>103</td>
</tr>
<tr>
<td><strong>2 safe problems</strong></td>
<td>154</td>
<td>112</td>
<td>138</td>
<td>85</td>
</tr>
<tr>
<td><strong>3 unsafe problems</strong></td>
<td>31</td>
<td>26</td>
<td>22</td>
<td>18</td>
</tr>
<tr>
<td><strong>4 incorrect answers</strong></td>
<td>0</td>
<td>9</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td><strong>5 false alarms</strong></td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>14</td>
</tr>
<tr>
<td><strong>6 missed bugs</strong></td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>7 errors</strong></td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td><strong>8 timed-out problems</strong></td>
<td>31</td>
<td>51</td>
<td>52</td>
<td>77</td>
</tr>
<tr>
<td><strong>9 total time</strong></td>
<td>10717.34</td>
<td>15788.21</td>
<td>15770.33</td>
<td>23259.19</td>
</tr>
<tr>
<td><strong>10 average time</strong></td>
<td>57.93</td>
<td>114.41</td>
<td>98.56</td>
<td>225.82</td>
</tr>
</tbody>
</table>
**Answer set Programming:**

Reduce the design of protocols to the computation of answer sets

- logic program $\Rightarrow$ encoding of a problem
- answer sets (model) $\Rightarrow$ solutions of a problem

$$\text{logic program } P \rightarrow \text{ASP System} \rightarrow \{ \text{AS} | \text{AS} \models P \}$$
Time dependant behavioural properties of Concurrent Programs:

- safety
- liveness

Specified in a Temporal Logic, i.e., Computation Tree Logic (CTL):

- path quantifiers: for all paths $A$, for some paths $E$
- temporal operators: eventually $F$, globally $G$, next $X$,....
Process structure: encoded as a function $f$

*either* the *identity function* $id$

$$
\begin{array}{cccccc}
P & \xrightarrow{id} & P & \xrightarrow{id} & \cdots & \xrightarrow{id} & P & \xrightarrow{id} & P \\
\uparrow & & & & & & & & \downarrow
\end{array}
$$

(Dijkstra’s semaphore)

*or* a generator of a cyclic group $\{id, f, \ldots, f^{k-1}\}$ of order $k$

$$
\begin{array}{cccccccc}
P_1 & \xrightarrow{f} & P_2 & \xrightarrow{f} & \cdots & \xrightarrow{f} & P_{k-1} & \xrightarrow{f} & P_k \\
\uparrow & & & & & & & & \downarrow
\end{array}
$$

(Peterson’s algorithm)
A 2-process protocol

Given the specification $\varphi$

Behavioural property: $\text{AG} \neg (x_1 = u \land x_2 = u)$ (mutual exclusion)

Structural property: $\begin{array}{c}
0 \\
1
\end{array} \begin{array}{c}
0 \\
1
\end{array}$

(A) encode it as ASP Program $P$

$$\text{false} : - \text{not ag(neg(and(local(p1,u),local(p2,u)))))}.$$  

(B) compute the answer sets of $P$

(C) decode the protocols from the answer sets

\[
\begin{align*}
\text{x}_1 & := t; \text{x}_2 := t; \text{y} := 0 \\
\text{P}_1 : & \quad \text{true} \rightarrow \text{if} \\
& \quad \text{x}_1 = t \land \text{y} = 0 \rightarrow \text{x}_1 := u; \text{y} := 0; \\
& \quad \text{|| x}_1 = t \land \text{y} = 0 \rightarrow \text{x}_1 := w; \text{y} := 1; \\
& \quad \text{fi} \\
\text{P}_2 : & \quad \text{true} \rightarrow \text{if} \\
& \quad \text{x}_2 = u \land \text{y} = 1 \rightarrow \text{x}_2 := t; \text{y} := 1; \\
& \quad \text{|| x}_2 = t \land \text{y} = 1 \rightarrow \text{x}_2 := w; \text{y} := 0; \\
& \quad \text{fi}
\end{align*}
\]
Complexity of the synthesis procedure

Theorem

For any number $k > 1$ of processes, for any symmetric program structure $\sigma$ over $L$ and $D$, and for any CTL formula $\varphi$, an answer set of the logic program $\Pi_\varphi \cup \Pi_\sigma$ can be computed in

(i) exponential time w.r.t. $k$,
(ii) linear time w.r.t. $|\varphi|$, and
(iii) nondeterministic polynomial time w.r.t. $|L|$ and w.r.t. $|D|$. 
Experimental results

Specification:
- Mutual Exclusion (ME)
- Starvation Freedom (SF)
- Bounded Overtaking (BO)
- Maximal Reactivity (MR)

Synthesized $k$-process concurrent programs:

| Program       | Satisfied Properties | $|ans(P)|$ | Time (sec) |
|---------------|----------------------|---------|------------|
| mutex for 2 processes | ME                  | 10      | 0.011      |
|               | ME                  | 10      | 0.012      |
|               | ME, SF              | 2       | 0.032      |
|               | ME, SF, BO          | 2       | 0.045      |
|               | ME, SF, BO, MR      | 2       | 0.139      |
| mutex for 3 processes | ME                  | 9       | 0.036      |
|               | ME                  | 14      | 0.036      |
|               | ME, SF              | 6       | 3.487      |
|               | ME, SF, BO          | 4       | 4.323      |
Conclusions

**Verification Framework**, which is parametric with respect to
- the language of the programs to be verified, and
- the logic of the property to be checked.

Instantiated to prove partial correctness of integer and array C programs
Implemented and available as a stand-alone system: the **VeriMAP** tool, which is competitive with respect to others CLP-based software model checkers.

**Synthesis Framework**, a fully declarative solution
- reduces the design of a concurrent program
to the design of its formal specification
- independent of the ASP solver