Program Verification using Constraint Handling Rules and Array Constraint Generalizations

Emanuele De Angelis\textsuperscript{1,3}, 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

CILC 2014
Torino, June 17, 2014
Given the program \textit{prog} and the formal specification \( \varphi \)

while\((x < n)\) {
    \(x = x + 1;\)
    \(y = y + 2;\)
}

\(\{ x = 0 \land y = 0 \land n \geq 1 \} \) \textit{prog} \( \{ y > x \} \)

(A) generate the \textbf{verification conditions} (VCs)

1. \(x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)\) \hspace{1cm} Initialization
2. \(P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n)\) \hspace{1cm} Loop invariant
3. \(P(x, y, n) \land x \geq n \rightarrow y > x\) \hspace{1cm} Exit

(B) prove they are \textbf{satisfiable}

If \textbf{satisfiable} then the correctness triple \textbf{hold}. 
VCs are **satisfiable** if there is an **interpretation** that makes them true. For instance, the interpretation

\[ P(x, y, n) \equiv (x=0 \land y=0 \land n\geq 1) \lor y > x \]

makes the VCs true

1'. \( x=0 \land y=0 \land n\geq 1 \rightarrow (x=0 \land y=0 \land n\geq 1) \lor y > x \)

2'. \( (x=0 \land y=0 \land n\geq 1) \lor y > x \) \land x < n

\( \rightarrow (x + 1 = 0 \land y + 2 = 0 \land n\geq 1) \lor y + 2 > x + 1 \)

3'. \( (x=0 \land y=0 \land n\geq 1) \lor y > x \) \land x \geq n \rightarrow y > x

and hence the triple \( \{x=0 \land y=0 \land n\geq 1\} \text{ prog } \{y > x\} \) holds.

How to prove the satisfiability of the VCs automatically?
Proving Satisfiability of Verification Conditions

The VCs are encoded as a **constraint logic program** \( V \)

1. \( p(X,Y,N) :- X=0, Y=0, N \geq 1. \) (Constrained fact)
2. \( p(X1,Y1,N) :- X < N, X1=X+1, Y1=Y+2, p(X,Y,N). \) (Rule)
3. \( \text{incorrect} :- X \geq N, Y \leq X, p(X,Y,N). \) (Query)

The VCs are satisfiable iff **incorrect** not in the **least model** of \( V \).

**Methods for proving the satisfiability of VCs:**
- CounterExample Guided Abstraction Refinement (CEGAR), Interpolation, Satisfiability Modulo Theories [Rybalchenko et al., McMillan, Alberti et al.]
- Symbolic execution of CLP [Jaffar et al.]
- Static Analysis and Transformation of CLP [Gallagher et al., Albert et al.]
A Transformation-based Method

Apply transformations that **preserve the least model** $M$ of $V$:

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

and derive the **equisatisfiable** $V'$:

5. $q(X_1,Y_1,N) :- X < N$, $X > Y$, $Y \geq 0$, $X_1 = X + 1$, $Y_1 = Y + 2$, $q(X,Y,N)$.
6. $\text{incorrect} :- X \geq N$, $X \geq Y$, $Y \geq 0$, $N \geq 1$, $q(X,Y,N)$.

i.e., $\text{incorrect} \in M(V)$ iff $\text{incorrect} \in M(V')$.

No constrained facts: $\text{incorrect} \notin M(V')$. 


How to transform $V$ into $V'$ automatically?

Some work done for programs over integers [PEPM-13].

Design automatic transformation strategies for programs over arrays.

- Verification method based on CLP program transformation
  - Semantics-preserving unfold/fold rules and strategies
  - VCs generation by specialization of CLP interpreters
    (semantics of the imperative language + proof rules)
  - VCs transformation by propagation of the property to be verified

- The verification method at work: Array Initialization
- Experimental evaluation
- Extending the verification framework
Given the specification \( \{ \varphi_{init} \} \) \( prog \) \( \{ \psi \} \) define \( \varphi_{error} \equiv \neg \psi \)

<table>
<thead>
<tr>
<th>Definition (The interpreter ( Int ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>incorrect :- errorConf(X), reach(X).</td>
</tr>
<tr>
<td>reach(X) :- tr(X,Y), reach(Y).</td>
</tr>
<tr>
<td>reach(X) :- initConf(X).</td>
</tr>
<tr>
<td>+ clauses for ( tr ) (the semantics of the programming language)</td>
</tr>
</tbody>
</table>

A program \( prog \) is \textbf{incorrect} w.r.t. \( \varphi_{init} \) and \( \varphi_{error} \) if from an initial configuration satisfying \( \varphi_{init} \) it is possible to reach a final configuration satisfying \( \varphi_{error} \). Otherwise, program \( prog \) is \textbf{correct}.

<table>
<thead>
<tr>
<th>Theorem</th>
</tr>
</thead>
<tbody>
<tr>
<td>( prog ) is correct iff ( \text{incorrect} \not\in M(Int) ) (the least model of ( Int ))</td>
</tr>
</tbody>
</table>
Running Example: Array Initialization

Given the program $\text{SeqInit}$ and the formal specification $\varphi$

$$
\begin{align*}
\text{i} &= 1; \\
\text{while}(i < n) \{ \\
    & \quad \text{a}[i] = \text{a}[i-1]+1; \\
    & \quad i = i + 1; \\
\} \\
\{ & i \geq 0 \land n = \text{dim}(a) \land n \geq 1 \\
\text{SeqInit} & \{ \forall j \ (0 \leq j \land j + 1 < n \rightarrow a[j] < a[j+1]) \}
\end{align*}
$$

CLP encoding of program SeqInit

A set of $\text{at}(\text{label}, \text{command})$ facts. while commands are replaced by $\text{ite}$ and $\text{goto}$. $\text{elem}(a, i)$ stands for $a[i]$.

$$
\begin{align*}
\text{at}(\ell_0, \text{asgn}(i, 1))_. \\
\text{at}(\ell_1, \text{ite}(\text{less}(i, n), \ell_2, \ell_h))_. \\
\text{at}(\ell_2, \text{asgn}(\text{elem}(a, i), \\
    \quad \text{plus}(\text{elem}(a, \text{minus}(i, 1)), 1)))_. \\
\text{at}(\ell_3, \text{asgn}(i, \text{plus}(i, 1)))_. \\
\text{at}(\ell_4, \text{goto}(\ell_1))_. \\
\text{at}(\ell_h, \text{halt})_.
\end{align*}
$$

CLP encoding of $\varphi_{\text{init}}$ and $\varphi_{\text{error}}$

$$
\begin{align*}
\text{initConf}(\ell_0, I, N, A) :&= I \geq 0, N \geq 1. \\
\text{errorConf}(\ell_h, N, A) :&= \\
    Z = W + 1, W \geq 0, W + 1 < N, U \geq V, \\
\end{align*}
$$
Program transformation is a technique that changes the syntax of a program and preserves its semantics.

Program Transformation of CLP can be used to
(A) generate the VCs
(B) prove the satisfiability of the VCs

 Interpreter: $Int$

 $Int$ w.r.t. $prog$ (generate the VCs)

 Verification Conditions: VCs

 Propagate $\varphi_{init}$ or $\varphi_{error}$

 (prove the satisfiability of the VCs)

 $prog$ correct
 (no constrained facts)

 $prog$ incorrect
 (the fact $incorrect$ in VCs)
Rule-based program transformation

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

- the transformation rules **preserve** the least model:

  \[
  \text{Theorem (Rules are semantics preserving)} \quad \text{incorrect} \in M(P) \iff \text{incorrect} \in M(\text{TransfP})
  \]

- the rules must be guided by a **strategy**.

[Burstall-Darlington 77, Tamaki-Sato 84, Etalle-Gabbrielli 96]
The Unfold/Fold Transformation strategy

Transform($P$)

$TransfP = \emptyset$;

Defs = \{incorrect :- errorConf(X), reach(X)\};

while $\exists q \in$ Defs do

% execute a symbolic evaluation step (resolution)
Cls = Unfold($q$);

% remove unsatisfiable and subsumed clauses
Cls = ClauseRemoval(Cls);

% introduce new predicates (e.g., a loop invariant)
Defs = (Defs $-$ \{q\}) $\cup$ Define(Cls);

% match a predicate definition
$TransfP = TransfP \cup$ Fold(Cls, Defs);

od
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 SeqInit (Verification Conditions)

```
new1(I1,N,B) :- 1 ≤ I, I < N, D = I-1, I1 = I+1, V = U+1, read(A,D,U), write(A,I,V,B), new1(I,N,A).
new1(I,N,A) :- I = 1, N ≥ 1.
```

- A constrained fact is present:
  we cannot conclude that the program is correct.
- The fact $\text{incorrect}$ is not present:
  we cannot conclude that the program is incorrect.
The Unfold/Fold Transformation strategy

Transform($P$)

$TransfP = \emptyset$;
Defs = \{incorrect :- errorConf(X), reach(X)\};

while $\exists q \in$ Defs do
  Cls = Unfold($q$);
  Cls = ConstraintReplacement(Cls);
  Cls = ClauseRemoval(Cls);
  Defs = (Defs $\setminus$ \{q\}) $\cup$ Define_array(Cls);
  $TransfP = TransfP \cup$ Fold(Cls, Defs);
od
If $\mathcal{A} \models \forall (c_0 \leftrightarrow (c_1 \vee \ldots \vee c_n))$, where $\mathcal{A}$ is the Theory of Arrays

Then replace

$$H :- c_0, d, G$$

by

$$H :- c_1, d, G, \ldots, H :- c_n, d, G$$

**Constraint Handling Rules** for Constraint Replacement:

**AC** Array Congruence (if $i=j$ then $a[i]=a[j]$)

\[
\text{read}(A1, I, X) \setminus \text{read}(A2, J, Y) \leftrightarrow A1 == A2, I = J \mid X = Y.
\]

**CAC** Contrapositive Array Congruence (if $a[i] \neq a[j]$ then $i \neq j$)

\[
\text{read}(A1, I, X), \text{read}(A2, J, Y) \Rightarrow A1 == A2, X <> Y \mid I <> J.
\]

**ROW** Read-Over-Write ($\{a[i]=v; z=a[j]\}$ if $i=j$ then $z=a[i]$)

\[
\text{write}(A1, I, X, A2) \setminus \text{read}(A3, J, Y) \leftrightarrow A2 == A3 \mid (I = J, X = Y); (I <> J, \text{read}(A1, J, Y)).
\]
new3(A,B,C) :- A = 2 + H, B - H \leq 3, E - H \leq 1, E \geq 1, B - H \geq 2, ...
   read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
   reach(J,B,N).

by applying the ROW rule we get:

new3(A,B,C) :- J = 1 + D, A = 2 + D, K = 1 + I, I < F, ...
   read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
   reach(J,B,N).

new3(A,B,C) :- J = 1 + D, A = 2 + D, K = 1 + I, I < F, ...
   read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
   reach(J,B,N).

by applying the ROW (again) and the AC rules we get:

new3(A,B,C) :- A = 1 + H, E = 1 + D, J = -1 + H, K = 1 + L, D - H \leq -2, H < B, ...
   read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
   reach(J,B,M).
new3(A,B,C) :- A=2+H, B-H ≤ 3, E-H ≤ 1, E ≥ 1, B-H ≥ 2, ...,
read(N,H,M), read(C,D,F), write(N,J,K,C), read(C,E,G),
reach(J,B,N).

by applying the ROW rule we get:

new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ...,
read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
reach(J,B,N).

by applying the ROW (again) and the AC rules we get:

new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H ≤ -2, H<B, ...
read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
reach(J,B,M).
Array Initialization

\[
\text{new3}(A,B,C) :- \ A=2+H, \ B-H \leq 3, \ E-H \leq 1, \ E \geq 1, \ B-H \geq 2, \ldots, \\
\text{read}(N,H,M), \ \text{read}(C,D,F), \ \text{write}(N,J,K,C), \ \text{read}(C,E,G), \\
\text{reach}(J,B,N).
\]

by applying the ROW rule we get:

\[
\text{new3}(A,B,C) :- \ J=1+D, \ A=2+D, \ K=1+I, \ I<F, \ldots, \ J=E, \ K=G, \\
\text{read}(C,D,F), \ \text{read}(N,D,I), \ \text{write}(N,J,K,C), \ \text{read}(C,E,G), \\
\text{reach}(J,B,N).
\]

\[
\text{new3}(A,B,C) :- \ J=1+D, \ A=2+D, \ K=1+I, \ I<F, \ldots, \ J<>E, \\
\text{read}(C,D,F), \ \text{read}(N,D,I), \ \text{write}(N,J,K,C), \ \text{read}(C,E,G), \\
\text{reach}(J,B,N).
\]

by applying the ROW (again) and the AC rules we get:

\[
\text{new3}(A,B,C) :- \ A=1+H, \ E=1+D, \ J=-1+H, \ K=1+L, \ D-H \leq -2, \ H<B, \ldots \\
\text{read}(N,E,G), \ \text{read}(N,D,F), \ \text{read}(N,J,L), \ \text{write}(N,H,K,C), \\
\text{reach}(J,B,M).
\]
Array Initialization

\[ \text{new3}(A,B,C) :\ - A=2+H, B-H \leq 3, E-H \leq 1, E \geq 1, B-H \geq 2, \ldots, \]
\[ \text{read}(N,H,M), \text{read}(C,D,F), \text{write}(N,J,K,C), \text{read}(C,E,G), \]
\[ \text{reach}(J,B,N). \]

- by applying the ROW rule we get:

\[ \text{new3}(A,B,C) :\ - J=1+D, A=2+D, K=1+I, I<F, \ldots, J=E, K=G, \]
\[ \text{read}(C,D,F), \text{read}(N,D,I), \text{write}(N,J,K,C), \text{read}(C,E,G), \]
\[ \text{reach}(J,B,N). \]

\[ \text{new3}(A,B,C) :\ - J=1+D, A=2+D, K=1+I, I<F, \ldots, J<>E, \]
\[ \text{read}(C,D,F), \text{read}(N,D,I), \text{write}(N,J,K,C), \text{read}(C,E,G), \]
\[ \text{reach}(J,B,N). \]

- by applying the ROW (again) and the AC rules we get:

\[ \text{new3}(A,B,C) :\ - A=1+H, E=1+D, J=-1+H, K=1+L, D-H \leq -2, H<B, \ldots \]
\[ \text{read}(N,E,G), \text{read}(N,D,F), \text{read}(N,J,L), \text{write}(N,H,K,C), \]
\[ \text{reach}(J,B,M). \]
Introduces new predicate definitions, i.e., program invariants, required to prove the property of interest.

Problem: transformation process may introduce an infinite number of definitions.

Use of generalization operators:

- to ensure the termination of the transformation,
- to generate program invariants,

... two somewhat conflicting requirements:

- efficiency, to introduce as few definitions as possible,
- precision, to prove as many properties as possible.
Definitions are arranged as a tree:

```
incorrect :- i, A
...
newp :- c, B  # ancestor definition
newq :- d, B  # candidate definition
newr :- g, B  # generalized definition
```

Generalization operators based on widening and convex-hull.
Array Constraint Generalizations

We decorate CLP variables with the variable identifiers of the imp. program.

\[
\text{incorrect} :: - i, A \\
\ldots \quad \text{newp} :- c, \text{read}, B \quad \text{ancestor definition} \\
\quad \text{newq} :- d, \text{read}, \text{read}, B \quad \text{candidate definition} \\
\quad \text{newr} :- g_1, \text{read}, B \quad \text{vs} \quad \text{newr} :- g_2, \text{read}, B
\]

The Specialized Interpreter for SeqInit (Verification Conditions)

\[
\text{incorrect} :: - Z=W+1, W \geq 0, W+1 < N, U \geq V, N \leq I, \\
\quad \text{read}(A, W_j, U^a[j]), \text{read}(A, Z_j^1, V^a[j^1]), \text{new1}(I, N, A). \\
\text{new1}(I1, N, B) :: - 1 \leq I, I < N, D=I-1, I_1=I+1, V=U+1, \\
\quad \text{read}(A, D^i, U^a[i]), \text{write}(A, I, V, B), \text{new1}(I, N, A). \\
\text{new1}(I, N, A) :: - I=1, N \geq 1.
\]
We decorate CLP variables with the variable identifiers of the imp. program.

\[
\text{incorrect} :- i, A \\
\text{\ldots} \\
\text{newp} :- c, \text{read}, B \\
\text{newq} :- d, \text{read}, \text{read}, B \\
\text{newr} :- g_1, \text{read}, B \\
\text{newr} :- g_2, \text{read}, B
\]

The Specialized Interpreter for SeqInit (Verification Conditions)

\[
\text{incorrect} :- Z=W+1, W \geq 0, W+1 < N, U \geq V, N \leq I, \\
\text{read}(A, W_j, U^{a[j]}), \text{read}(A, Z^{j_1}, V^{a[j_1]}), \text{new1}(I, N, A). \\
\text{new1}(I_1, N, B) :- 1 \leq I, I < N, D = I - 1, I_1 = I + 1, V = U + 1, \\
\text{read}(A, D_i, U^{a[i]}), \text{write}(A, I, V, B), \text{new1}(I, N, A). \\
\text{new1}(I, N, A) :- I = 1, N \geq 1.
\]
ancestor definition:

\[
\text{new3}(I, N, A) : - \ E + 1 = F, E \geq 0, I > F, G \geq H, N > F, N \leq I + 1, \\
\text{read}(A, E_j, G^a[j]), \text{read}(A, F^{j1}_1, H^a[j1]), \text{reach}(I, N, A).
\]

candidate definition:

\[
\text{new4}(I, N, A) : - \ E + 1 = F, E \geq 0, I > F, G \geq H, I = 1 + I_1, I_1 + 2 \leq C, N \leq I_1 + 3, \\
\text{read}(A, E_j, G^a[j]), \text{read}(A, F^{j1}_1, H^a[j1]), \text{read}(A, P^{i_1}, Q^a[i]), \\
\text{reach}(I, N, A).
\]

generalized definition:

\[
\text{new5}(I, N, A) : - \ E + 1 = F, E \geq 0, I > F, G \geq H, N > F, \\
\text{read}(A, E_j, G^a[j]), \text{read}(A, F^{j1}_1, H^a[j1]), \text{reach}(I, N, A).
\]

in the paper: any variable of the form $G^v$ is encoded by a constraint $\text{val}(v, G)$
The VeriMAP tool http://map.uniroma2.it/VeriMAP
### Experimental evaluation

<table>
<thead>
<tr>
<th>Program</th>
<th>$Gen_{W,I,n}$</th>
<th>$Gen_{H,V,\subseteq}$</th>
<th>$Gen_{H,V,n}$</th>
<th>$Gen_{H,I,\subseteq}$</th>
<th>$Gen_{H,I,n}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubblesort-inner</td>
<td>0.9</td>
<td>unknown</td>
<td>unknown</td>
<td>unknown</td>
<td>1.52</td>
</tr>
<tr>
<td>copy-partial</td>
<td>unknown</td>
<td>unknown</td>
<td>3.52</td>
<td>3.51</td>
<td>3.54</td>
</tr>
<tr>
<td>copy-reverse</td>
<td>unknown</td>
<td>unknown</td>
<td>5.25</td>
<td>unknown</td>
<td>5.23</td>
</tr>
<tr>
<td>copy</td>
<td>unknown</td>
<td>unknown</td>
<td>5.00</td>
<td>4.88</td>
<td>4.90</td>
</tr>
<tr>
<td>find-first-non-null</td>
<td>0.14</td>
<td>0.66</td>
<td>0.64</td>
<td>0.28</td>
<td>0.27</td>
</tr>
<tr>
<td>find</td>
<td>1.04</td>
<td>6.53</td>
<td>2.35</td>
<td>2.33</td>
<td>2.29</td>
</tr>
<tr>
<td>first-not-null</td>
<td>0.11</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>init-backward</td>
<td>unknown</td>
<td>1.04</td>
<td>1.04</td>
<td>1.03</td>
<td>1.04</td>
</tr>
<tr>
<td>init-non-constant</td>
<td>unknown</td>
<td>2.51</td>
<td>2.51</td>
<td>2.47</td>
<td>2.47</td>
</tr>
<tr>
<td>init-partial</td>
<td>unknown</td>
<td>0.9</td>
<td>0.89</td>
<td>0.9</td>
<td>0.89</td>
</tr>
<tr>
<td>init-sequence</td>
<td>unknown</td>
<td>4.38</td>
<td>4.33</td>
<td>4.41</td>
<td>4.29</td>
</tr>
<tr>
<td>init</td>
<td>unknown</td>
<td>1.00</td>
<td>0.97</td>
<td>0.98</td>
<td>0.98</td>
</tr>
<tr>
<td>insertionsort-inner</td>
<td>0.58</td>
<td>2.41</td>
<td>2.4</td>
<td>2.38</td>
<td>2.37</td>
</tr>
<tr>
<td>max</td>
<td>unknown</td>
<td>unknown</td>
<td>0.8</td>
<td>0.81</td>
<td>0.82</td>
</tr>
<tr>
<td>partition</td>
<td>0.84</td>
<td>1.77</td>
<td>1.78</td>
<td>1.76</td>
<td>1.76</td>
</tr>
<tr>
<td>rearrange-in-situ</td>
<td>unknown</td>
<td>unknown</td>
<td>3.06</td>
<td>3.01</td>
<td>3.03</td>
</tr>
<tr>
<td>selectionsort-inner</td>
<td>unknown</td>
<td>time-out</td>
<td>unknown</td>
<td>2.84</td>
<td>2.83</td>
</tr>
<tr>
<td>precision</td>
<td>6</td>
<td>10</td>
<td>15</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>total time</td>
<td>3.61</td>
<td>21.42</td>
<td>34.76</td>
<td>31.81</td>
<td>38.45</td>
</tr>
<tr>
<td>average time</td>
<td>0.60</td>
<td>2.14</td>
<td>2.31</td>
<td>2.12</td>
<td>2.26</td>
</tr>
</tbody>
</table>
Conclusions and Future Work

- Parametric verification framework (semantics and logic, constraint domain)
  - CLP as a metalanguage
  - agile way of synthesizing software verifiers (Rybalchenko et al.)
- Semantics preserving transformations
  - iteration, incremental verification
  - use Horn clauses for passing information between verifiers (McMillan)
- Future work
  - more experiments (e.g., nested loops)
  - more theories (lists, heaps, etc.)
  - Other programming languages, properties, proof rules