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”, Italy
\textsuperscript{2}University of Rome “Tor Vergata”, Italy
\textsuperscript{3}CNR - Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

Workshop on:
\textbf{Metodi dichiarativi nella verifica di sistemi parametrici}

Milano, September 25–26, 2014
Outline

- **Encoding partial correctness of array programs** into CLP programs.

  - **(A) Generation of the verification conditions** (i.e., removal of the interpreter).

  - **(B) Check of satisfiability of the verification conditions** via CLP program transformation.

- **Manipulation of Array Constraints** via Constraint Handling Rules (CHR).

- **Experimental evaluation**.
Proving partial correctness: our method
Consider a program and a partial correctness triple:

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

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

\[
\{ y > x \}
\]

(A) Generate the Verification Conditions (VC’s)

1. \( x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n) \) Initialization
2. \( P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n) \) Loop
3. \( P(x, y, n) \land x \geq n \land y \leq x \rightarrow \text{incorrect} \) Exit

(B) If the VC’s are satisfiable (i.e., there is an interpretation for \( P(x, y, n) \) that makes 1, 2, and 3 true), then the partial correctness triple holds.
The CLP Transformation Method

(A) Generate the VC’s as a CLP program from the partial correctness triple and the definition of the semantics:

\[ V: \]
1*. \( p(X,Y,N) : - X=0, \ Y=0, \ N \geq 1. \) (a constrained fact)
2*. \( p(X_1,Y_1,N) : - X < N, \ X_1=X+1, \ Y_1=Y+2, \ p(X,Y,N). \)
3*. \( \text{incorrect} : - X \geq N, Y \leq X, \ p(X,Y,N). \)

Theorem: The VC’s are satisfiable iff incorrect \( \not\in \) the least model \( M(V). \)

(B) Apply transformation rules that preserve the least model \( M(V). \)

\[ V': \]
4. \( q(X_1,Y_1,N) : - X < N, X > Y, Y \geq 0, \ X_1=X+1, \ Y_1=Y+2, \ q(X,Y,N). \)
5. \( \text{incorrect} : - X \geq N, Y \leq X, Y \geq 0 \ N \geq 1, \ q(X,Y,N). \)

least model preserved: incorrect \( \not\in M(V) \) iff incorrect \( \not\in M(V') \)
no constrained facts for q: incorrect \( \not\in M(V') \)
Thus,
\[ \{ x=0 \land y=0 \land n \geq 1 \} \ prog \ \{ y > x \} \ \text{holds.} \]
Encoding partial correctness of array programs into CLP
Consider the triple \( \{ \varphi_{init} \} \ prog \ \{ \neg \varphi_{error} \} \).

A program \( prog \) is incorrect w.r.t. \( \varphi_{init} \) and \( \varphi_{error} \) if a final configuration satisfying \( \varphi_{error} \) is reachable from an initial configuration satisfying \( \varphi_{init} \).

**Definition (the interpreter \( Int \) with the transition predicate \( tr(X,Y) \))**

reach(X) :- initConf(X).
reach(Y) :- tr(X,Y), reach(X).
incorrect :- errorConf(X), reach(X).

+ clauses for \( tr \) (i.e., the operat. semantics of the programming language)

**Theorem**

\( prog \) is incorrect iff \( \text{incorrect} \in M(\text{Int}) \)

A program \( prog \) is correct iff it is not incorrect.
### tr(X,Y): the operational semantics

| L: Id = Expr | \( \text{tr}( \text{cf(cmd(L,asgn(Id,Expr)),S)}, \text{cf(cmd(L1,C1),S1)}) \):-
| | \( \text{aeval(Expr,S,V)}, \) **evaluate expression**
| | \( \text{update(Id,V,S,S1)}, \) **update store**
| | \( \text{nextlabel(L,L1)}, \) **next label**
| | \( \text{at(L1,C1)}). \) **next command** |
| L: if(Expr){ | \( \text{tr}( \text{cf(cmd(L,ite(Expr,L1,L2)),S)}, \text{cf(C,S)}) \):-
| | \( \text{beval(Expr,S)}, \) **expression is true**
| | \( \text{at(L1,C)}). \) **next command**
| | else | \( \text{tr}( \text{cf(cmd(L,ite(Expr,L1,L2)),S)}, \text{cf(C,S)}) \):-
| | \( \text{beval(not(Expr),S)}, \) **expression is false**
| | \( \text{at(L2,C)}). \) **next command**
| L: goto L1 | \( \text{tr}( \text{cf(cmd(L,goto(L1)),S)}, \text{cf(C,S)}) \):-
| | \( \text{at(L1,C)}). \) **next command**
tr(X,Y): the operational semantics for array assignment

**array assignment**: \( L : a[ie] = e \)

**old store**: \( S \)

**new store**: \( S_1 \)

**transition**:

\[
\text{tr}( \text{cf(cmd}(L, \text{asgn}(\text{elem}(A,IE), E)), S), \text{cf}(\text{cmd}(L1, C), S1)) :- \\
\text{eval}(IE, S, I), \\
\text{eval}(E, S, V), \\
\text{lookup}(S, \text{array}(A), FA), \\
\text{write}(FA, I, V, FA1), \\
\text{update}(S, \text{array}(A), FA1, S1), \\
\text{nextlab}(L, L1), \\
\text{at}(L1, C).
\]

- **old configuration** \( \text{cf} \)
- **new configuration** \( \text{cf} \)
- **evaluate index expr** \( IE \)
- **evaluate expression** \( E \)
- **get array FA from store** \( S \)
- **update array FA, getting** \( FA1 \)
- **update store S, getting** \( S1 \)
- **next label** \( L1 \)
- **command C at next label**
Program \textit{UpInit}

\begin{verbatim}
i = 1;
while (i < n) {
a[i] = a[i-1]+1;
i = i+1;
}
\end{verbatim}

An Execution of \textit{UpInit} (assume $n=4$ and $a[0]=2$)

\begin{align*}
[2, ?, ?, ?] & \rightarrow [2, 3, ?, ?] \rightarrow [2, 3, 4, ?] \rightarrow [2, 3, 4, 5]
\end{align*}
Given the **program** *UpInit* and the **partial correctness triple**

\[
i = 1; \\
\text{while } (i < n) \{ \\
a[i] = a[i - 1] + 1; \\
i = i + 1; \\
\} \\
\{ i \geq 0 \land n \geq 1 \land n = \text{dim}(a) \} \\
\text{UpInit} \\
\{ \forall j \ (0 \leq j \land j + 1 < n \rightarrow a[j] < a[j + 1]) \} \]

**CLP encoding of program *UpInit***

- A set of *at*(label, command) facts.
- **while** becomes *ite* + *goto*.
- *a[i]* becomes *elem*(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 \( \phi_{\text{init}} \) and \( \phi_{\text{error}} \)**

\[
\begin{align*}
\text{initConf}(\ell_0, I, N, A) :&= \\
I \geq 0, N \geq 1. \\
\text{errorConf}(\ell_h, N, A) :&= \\
W \geq 0, W + 1 < N, \quad Z = W + 1, U \geq V, \\
\text{read}(A, [W, U]), \text{read}(A, [Z, V]).
\end{align*}
\]
The array constraints: read and write

• if $a[i] = v$, then $\text{read}(A, I, V)$ holds

• if $a[i] := v$, then $\text{write}(A, I, V, B)$ holds, that is, the array $B$ is an array identical to $A$ except that array $B$ in position $I$ has value $V$
The Transformation-based Verification Method

**Interpretation:** \( Int \)

(A) **Specialize** \( Int \) w.r.t. \( prog \) (i.e., removal of the interpreter)

**Verification Conditions:** \( VC's \)

(B) **Transformation by propagation of the constraints**
derived from \( \varphi_{init} \) or \( \varphi_{error} \)

- \( prog \) **correct** if no constrained facts appear in the \( VC's \).
- \( prog \) **incorrect** if the fact \( \text{incorrect} \) appears in the \( VC's \).
Transform

\[ TransfP = \emptyset; \]
\[ \text{Defs} = \{ \text{incorrect} :- \text{errorConf}(X), \text{reach}(X) \}; \]
\[ \text{while } \exists q \in \text{Defs} \ \text{do} \]
\[ \text{Cls} = \text{Unfolding}(q); \]
\[ \text{Cls} = \text{ConstraintReplacement}(\text{Cls}); \]
\[ \text{Cls} = \text{ClauseRemoval}(\text{Cls}); \]
\[ \text{Defs} = (\text{Defs} - \{ q \}) \cup \text{Definition}_{\text{array}}(\text{Cls}); \]
\[ TransfP = TransfP \cup \text{Folding}(\text{Cls}, \text{Defs}); \]
\[ \text{od} \]
The specialization of $Int$ w.r.t. $prog$ removes all references to:

- $tr$ and
- $at$

**VC: The Verification Conditions for UpInit**

```
incorrect :- Z=W+1, W≥0, W+1<N, U≥V, N≤I,
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 **incorrect** is not present: we cannot conclude that the program is **incorrect** either.
Check satisfiability of VC’s via CLP transformation:
Propagation of Integer and Array Constraints
Constraint Replacement Rules (CHR)

If $\mathcal{A} \models \forall (c_0 \leftrightarrow (c_1 \lor \ldots \lor c_n))$, 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 [Frühwirth et al.] for Constraint Replacement:

AC1. Array-Congruence-1: if $i=j$ then $a[i]=a[j]$
    read($A, I, X$) \ read($A_1, J, Y$) $\Leftrightarrow$ $A == A_1, I = J$ | $X = Y$.

AC2. Array-Congruence-2: if $a[i] \neq a[j]$ then $i \neq j$
    read($A, I, X$), read($A_1, J, Y$) $\Rightarrow$ $A == A_1, X <> Y$ | $I <> J$.

ROW. Read-Over-Write: \{a[i]=x; y=a[j]\} if $i=j$ then $x=y$
    write($A, I, X, A_1$) \ read($A_2, J, Y$) $\Leftrightarrow A_1 == A_2$ |
    ($I = J, X = Y$) ; ($I <> J, \text{read}(A,J,Y)$).
new3(A,B,C) :- A=2+H, B-H \leq 3, E-H \leq 1, E \geq 1, B-H \geq 2, \ldots,
   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:

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

- by applying the ROW, AC1, and AC2 rules:

new3(A,B,C) :- A=1+H, E=1+D, J=-1+H, K=1+L, D-H \leq -2, H<B, \ldots
   read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C),
   reach(J,B,M).
Definitions are arranged as a tree:

```
incorrect :- errConf, A
...
newp :- c, B
  
  newq :- d, B

newr :- g, B
```

: ancestor definition
: candidate definition
: \textit{generalized} definition

$d \rightarrow g$

Generalization operators based on \textit{widening} and \textit{convex-hull} [Cousot-Cousot 77, Cousot-Halbwachs 78].
We decorate CLP variables with the associated identifiers of the imperative program.

**VC: The Verification Conditions for UpInit (decorated)**

\[
\text{incorrect} :- Z=W+1, \ W \geq 0, \ W+1 < N, \ U \geq V, \ N \leq I, \\
\text{read}(A,W_j,U[a][j]), \ \text{read}(A,Z_{j1},V[a][j1]), \ \text{new1}(I,N,A).
\]

\[
\text{new1}(I_1,N,B) :- 1 \leq I, \ I < N, \ D=I-1, \ I_1=I+1, \ V=U+1, \\
\text{read}(A,D_i,U[a][i]), \ \text{write}(A,I,V,B), \ \text{new1}(I,N,A).
\]

\[
\text{new1}(I,N,A) :- I=1, \ N \geq 1.
\]
Up Array Initialization

\[ \text{new3}(I,N,A) : - \ E + 1 = F, \ E \geq 0, \ I > F, \ G \geq H, \ N > F, \ N \leq I + 1, \]
\[ \text{read}(A, E^j, G^a[j]), \text{read}(A, F_1^j, H^a[j_1]), \text{reach}(I, N, A). \]

\[ \text{new4}(I,N,A) : - \ E + 1 = F, \ E \geq 0, \ I > F, \ G \geq H, \ I = 1 + I_1, \ I_1 + 2 \leq C, \ N \leq I_1 + 3, \]
\[ \text{read}(A, E^j, G^a[j]), \text{read}(A, F_1^j, H^a[j_1]), \text{read}(A, P^i, Q^a[i]), \]
\[ \text{reach}(I, N, A). \]

\[ \text{new5}(I,N,A) : - \ E + 1 = F, \ E \geq 0, \ I > F, \ G \geq H, \ N > F, \]
\[ \text{read}(A, E^j, G^a[j]), \text{read}(A, F_1^j, H^a[j_1]), \text{reach}(I, N, A). \]

In the paper: a variable of the form \( G^v \) is encoded by \( \text{val}(v,G) \).
By applying the transformation strategy *Transform* to the verification conditions for *UpInit*, we get:

**VC': Transformed verification conditions for *UpInit***

\[
\text{incorrect} : - J1 = J + 1, J \geq 0, J1 < I, AJ \geq AJ1, D = I - 1, N = I + 1, Y = X + 1, \\
\text{read}(A, J, AJ), \text{read}(A, J1, AJ1), \text{read}(A, D, X), \text{write}(A, I, Y, B), \\
\text{new1}(I, N, A).
\]

\[
\text{new1}(I1, N, B) : - I1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, N \leq I + 2, \\
I \geq 1, Z < I, Z \geq 1, N > I, U \geq V, \text{read}(A, W, U), \text{read}(A, Z, V), \\
\text{read}(A, D, X), \text{write}(A, I, Y, B), \text{new5}(I, N, A).
\]

\[
\text{new5}(I1, N, B) : - I1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, I \geq 1, \\
Z < I, Z \geq 1, N > I, U \geq V, \text{read}(A, W, U), \text{read}(A, Z, V), \\
\text{read}(A, D, X), \text{write}(A, I, Y, B), \text{new5}(I, N, A).
\]

No constrained facts in \( VC' \): \( \text{incorrect} \not\in M(VC') \).
The program *UpInit* is correct.
Experimental results
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></td>
<td></td>
</tr>
<tr>
<td>verified</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>
Future Work

- Proving recursively defined properties
- Imperative programs with recursive functions
- More data structure theories (lists, heaps, etc.)
- Other programming languages, properties, and proof rules
Proving recursively defined properties
The GCD program

\[ x = m; \quad y = n; \]

while \((x \neq y)\) {
  if \((x > y)\) \(x = x - y;\)
  else \(y = y - x;\)
}

\[ z = x; \]

\% \(z = \text{greatest-common-divisor}\)
\% of \(m\) and \(n\)

partial correctness triple

\[ \varphi_{\text{init}}(m,n) \equiv \{ m \geq 1 \land n \geq 1 \} \]

\[ \varphi_{\text{error}}(m,n,z) \equiv \{ \exists d (\gcd(m,n,d) \land d \neq z) \} \]

GCD property

\[ \gcd(X, Y, D) : - \quad X > Y, \quad X1 = X - Y, \quad \gcd(X1, Y, D). \]
\[ \gcd(X, Y, D) : - \quad X < Y, \quad Y1 = Y - X, \quad \gcd(X, Y1, D). \]
\[ \gcd(X, Y, D) : - \quad X = Y, \quad Y = D. \]
Proving recursively defined properties

**CLP encoding of GCD**

reach(X) :- initConf(X).
reach(Y) :- tr(X,Y), reach(X).
incorrect :- errorConf(X), reach(X).

initConf(cf(cmd(0, asgn(int(x), int(m)))),
         [[int(m), M], [int(n), N], [int(x), X], [int(y), Y], [int(z), Z]]) :-
        M ≥ 1, N ≥ 1. % φ_{init}(m,n)

errorConf(cf(cmd(h, halt),
          [[int(m), M], [int(n), N], [int(x), X], [int(y), Y], [int(z), Z]]) :-
        gcd(M, N, D), D ≠ Z. % φ_{error}(m,n,z)

Generation of VC’s; Propagation of φ_{error}(m,n,z)

**Transformed GCD**

incorrect :- M ≥ 1, N ≥ 1, M > N, X1 = M−N, Z ≠ D, new1(M, N, X1, N, Z, D).
incorrect :- M ≥ 1, N ≥ 1, M < N, Y1 = N−M, Z ≠ D, new1(M, N, M, Y1, Z, D).
new1(M, N, X, Y, Z, D) :- M ≥ 1, N ≥ 1, X > Y, X1 = X−Y, Z ≠ D, new1(M, N, X1, Y, Z).
new1(M, N, X, Y, Z, D) :- M ≥ 1, N ≥ 1, X < Y, Y1 = Y−X, Z ≠ D, new1(M, N, X, Y1, Z).

No constrained fact: the GCD program is correct.
Try the VeriMAP tool!

http://map.uniroma2.it/VeriMAP
Why Use CLP Transformation for Verification?

- CLP transformation can be used both for generating VC’s and for proving their satisfiability.

- CLP transformation is parametric with respect to:
  - the programming language and its semantics
  - the properties to be proved
  - the proof rules
  - the theories of the data structures

- The input CLP program and the transformed CLP program are semantically equivalent. This allows:
  - composition of transformations
  - incremental verification of properties
  - easy inter-operability with other verifiers that use Horn-clause format.
Conclusions

Our verification framework:

- CLP as a metalanguage for a formal definition of the programming language semantics and program properties

- Semantics preserving transformations of CLP as proof rules which are programming language independent.
Automatic Proofs of Satisfiability of VC’s

Various methods (incomplete list):

- Verification of safety of infinite state systems in Constraint Logic Programming (CLP) [Delzanno-Podelski]

- CounterExample Guided Abstraction Refinement (CEGAR), Interpolation, Satisfiability Modulo Theories [Podelski-Rybalchenko, Bjørner, McMillan, Alberti et al.]

- Symbolic execution of Constraint Logic Programs [Jaffar et al.]

- Static Analysis and Transformation of Constraint Logic Programs [Gallagher et al., Albert et al., De Angelis et al.]