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.
- Generation of the verification conditions (i.e., removal of the interpreter).
- Check of satisfiability of the verification conditions via CLP program transformation.
- Manipulation of Integer and Array Constraints via Constraint Handling Rules (CHR).
- Experimental evaluation.
Proving partial correctness of imperative programs through transformation of CLP programs
Consider a program and a partial correctness triple:

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

(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 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 formal definition of the semantics:

\[ V: \]

\[ V^*: \]

\[ p(X,Y,N) : \neg X = 0, \ Y = 0, \ N \geq 1. \]  

\[ p(X1,Y1,N) : \neg X < N, \ X1 = X + 1, \ Y1 = Y + 2, \ p(X, Y, N). \]  

\[ incorrect : \neg X \geq N, \ Y \leq X, \ p(X, Y, N). \]  

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

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

\[ V': \]

\[ q(X1, Y1, N) : \neg X < N, \ X > Y, \ Y \geq 0, \ X1 = X + 1, \ Y1 = Y + 2, \ q(X, Y, N). \]  

\[ incorrect : \neg X \geq N, \ Y \leq X, \ Y \geq 0 \ N \geq 1, \ q(X, Y, N). \]  

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,

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

holds.
Encoding partial correctness of array programs into CLP
Consider the triple \( \{ \varphi_{\text{init}} \} \ prog \ \{ \neg \varphi_{\text{error}} \} \).

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

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

\[
\begin{align*}
\text{reach}(X) & :- \ initConf(X). \\
\text{reach}(Y) & :- \text{tr}(X,Y), \text{reach}(X). \\
\text{incorrect} & :- \ text{errorConf}(X), \text{reach}(X). \\
\text{+ clauses for \text{tr} (i.e., the operat. semantics of the programming language)}
\end{align*}
\]

**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

<table>
<thead>
<tr>
<th>L: Id = Expr</th>
<th>tr( cf(cmd(L, asgn(Id, Expr)), S), cf(cmd(L1, C1), S1)) :- aeval(Expr, S, V), update(Id, V, S, S1), nextlabel(L, L1), at(L1, C1).</th>
</tr>
</thead>
<tbody>
<tr>
<td>L: if(Expr){</td>
<td>tr( cf(cmd(L, ite(Expr, L1, L2)), S), cf(C, S)) :- beval(Expr, S), at(L1, C).</td>
</tr>
<tr>
<td>L1: ... }</td>
<td>expression is true next command</td>
</tr>
<tr>
<td>else</td>
<td>tr( cf(cmd(L, ite(Expr, L1, L2)), S), cf(C, S)) :- beval(not(Expr), S), at(L2, C).</td>
</tr>
<tr>
<td>L2: ... }</td>
<td>expression is false next command</td>
</tr>
<tr>
<td>L: goto L1</td>
<td>tr( cf(cmd(L, goto(L1)), S), cf(C, S)) :- at(L1, C).</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
array assignment: $L : a[ie] = e$

old store: $S$

new store: $S_1$

transition:

$$\text{tr} \left( \text{cf}(\text{cmd}(L, \text{asgn}(\text{elem}(A, IE), E)), S), \text{cf}(\text{cmd}(L_1, C), S_1) \right) :$$

- $\text{eval}(IE, S, I)$,
- $\text{eval}(E, S, V)$,
- $\text{lookup}(S, \text{array}(A), FA)$,
- $\text{write}(FA, I, V, FA_1)$,
- $\text{update}(S, \text{array}(A), FA_1, S_1)$,
- $\text{nextlab}(L, L_1)$,
- $\text{at}(L_1, C)$.

old configuration $\text{cf}$

new configuration $\text{cf}$

evaluate index expr $IE$

evaluate expression $E$

get array $FA$ from store

update array $FA$, getting $FA_1$

update store $S$, getting $S_1$

next label $L_1$

command $C$ at next label
Running Example: Up Array Initialization

Program $UpInit$

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

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

\[
[2, ?, ?, ?] \rightarrow [2, 3, ?, ?] \rightarrow [2, 3, 4, ?] \rightarrow [2, 3, 4, 5]
\]
Running Example: Up Array Initialization

Given the program `UpInit` and the partial correctness triple

```plaintext
i=1;
while(i<n) {
    a[i]=a[i-1]+1;
    i=i+1;
}

{i ≥ 0 ∧ n ≥ 1 ∧ n = dim(a)}

UpInit

{∀j (0 ≤ j ∧ j + 1 < n → 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)`.

```plaintext
at(ℓ₀, asgn(i, 1)).
at(ℓ₁, ite(less(i, n), ℓ₂, ℓ₇)).
at(ℓ₂, asgn(elem(a, i),
    plus(elem(a, minus(i, 1)), 1))).
at(ℓ₃, asgn(i, plus(i, 1))).
at(ℓ₄, goto(ℓ₁)).
at(ℓ₇, halt).
```

CLP encoding of `ϕ_init` and `ϕ_error`

```plaintext
initConf(ℓ₀, I, N, A) :-
    I ≥ 0, N ≥ 1.

errorConf(ℓ₇, N, A) :-
    W ≥ 0, W + 1 < N, Z = W + 1, U ≥ V,
```
Generating Verification Conditions via CLP transformation
The Strategy for Generation (Specialization of Int)

Specialize($P$)

$TransfP = \emptyset$;

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

while $\exists q \in Defs$ do

% execute a symbolic evaluation step (i.e., resolution)
$Cls = \text{Unfolding}(q)$;

% remove unsatisfiable and subsumed clauses
$Cls = \text{ClauseRemoval}(Cls)$;

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

% match a predicate definition
$TransfP = TransfP \cup \text{Folding}(Cls, Defs)$;

od
The specialization of \textit{Int} w.r.t. \textit{prog} removes all references to:

- \textit{tr} and
- \textit{at}

\textbf{VC: The Verification Conditions for UpInit}

\begin{verbatim}
incorrect :- Z=W+1, W\geq 0, W+1<N, U\geq V, N\leq I,
new1(I1,N,B) :- 1\leq 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\geq 1.
\end{verbatim}

- A constrained fact is present:
  we cannot conclude that the program is \textit{correct}.
- The fact \textit{incorrect} is not present:
  we cannot conclude that the program is \textit{incorrect} either.
The Transformation-based Verification Method

Interpreter: \( Int \)

Specialize \( Int \) w.r.t. \( prog \) (removal of the interpreter)

Verification Conditions: \( VCs \)

Propagate \( \varphi_{\text{init}} \) or \( \varphi_{\text{error}} \)

\( prog \) correct if no constrained facts appear in the VCs.

\( prog \) incorrect if the fact \( \text{incorrect} \). appears in the VCs.
Checking satisfiability of VC’s via CLP transformation
The Strategy for Satisfiability

Transform($P$)

$$
TransfP = \emptyset; \\
Defs = \{\text{incorrect} :- \text{errorConf}(X), \text{reach}(X)\}; \\
\textbf{while } \exists q \in Defs \textbf{ do} \\
\quad Cls = \text{Unfolding}(q); \\
\quad Cls = \text{ConstraintReplacement}(Cls) ; \\
\quad Cls = \text{ClauseRemoval}(Cls); \\
\quad Defs = (Defs \setminus \{q\}) \cup \text{Definition}_{array}(Cls) ; \\
\quad TransfP = TransfP \cup \text{Folding}(Cls, Defs); \\
\textbf{od}
$$
Constraint manipulation in the theory of arrays
Constraint Replacement Rules (CHR’s)

If \( A \models \forall (c_0 \leftrightarrow (c_1 \lor \ldots \lor c_n)) \), where \( A \) 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(\( A_1, J, Y \)) \iff A == A_1, I = J \ \land \ 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 \ \lor \ 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(\( A_2, J, Y \)) \iff A_1 == A_2 \ \land \ (I = J, X = Y) ; (I <> J, read(\( A, J, Y \))).
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:

new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., 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, ..., J<>E, read(C,D,F), read(N,D,I), write(N,J,K,C), write(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≤-2, H<B, ..., read(N,E,G), read(N,D,F), read(N,J,L), write(N,H,K,C), reach(J,B,M).
Introduction of suitable new predicate definitions (they correspond to program invariants).

**Difficulty**: Introduction of an unbounded number of new predicate definitions.

**Solution**: Use of generalization operators:
- to ensure the termination of the transformation,
- to generate program invariants.
Definitions are arranged as a tree:

```
incorrect :- err, A

... newp :- c, B : ancestor definition

... newq :- d, B : candidate definition

newr :- g, B : generalized definition
```

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

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

Incorrect:

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

New1:

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

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

: ancestor definition

new3(I,N,A) :- E+1=F, E≥0, I>F, G≥H, N>F, N≤I+1,
read(A,Ej,G^a[j]), read(A,Fj1,H^a[j1]), reach(I,N,A).

: candidate definition

new4(I,N,A) :- E+1=F, E≥0, I>F, G≥H, I=1+I1, I1+2≤C, N≤I1+3,
read(A,Ej,G^a[j]), read(A,Fj1,H^a[j1]), read(A,P^i,Q^a[i]),
reach(I,N,A).

: generalized definition

new5(I,N,A) :- E+1=F, E≥0, I>F, G≥H, N>F,
read(A,Ej,G^a[j]), read(A,Fj1,H^a[j1]), reach(I,N,A).

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

\[ VC': \text{Transformed verification conditions for } UpInit \]

\[
\text{incorrect} : - J_1 = J + 1, J \geq 0, J_1 < I, AJ \geq AJ_1, D = I - 1, N = I + 1, Y = X + 1, \\
\text{read}(A, J, AJ), \text{read}(A, J_1, AJ_1), \text{read}(A, D, X), \text{write}(A, I, Y, B), \\
\text{new1}(I, N, A).
\]

\[
\text{new1}(I_1, N, B) : - I_1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, N \leq I + 2, \\
I \geq 1, Z < I, Z \geq 1, N > I, U \geq V, \text{read}(A, W, U), \text{read}(A, Z, V), \\
\text{read}(A, D, X), \text{write}(A, I, Y, B), \text{new5}(I, N, A).
\]

\[
\text{new5}(I_1, N, B) : - I_1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, I \geq 1, \\
Z < I, Z \geq 1, N > I, U \geq V, \text{read}(A, W, U), \text{read}(A, Z, V), \\
\text{read}(A, D, X), \text{write}(A, I, Y, B), \text{new5}(I, N, A).
\]

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
<table>
<thead>
<tr>
<th>Program</th>
<th>$Gen_{W,I,\mathbb{m}}$</th>
<th>$Gen_{H,V,\subseteq}$</th>
<th>$Gen_{H,V,\mathbb{m}}$</th>
<th>$Gen_{H,I,\subseteq}$</th>
<th>$Gen_{H,I,\mathbb{m}}$</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>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 \textit{GCD} program
\begin{verbatim}
x = m;  y = n;
while (x != y) {
    if (x > y) x = x - y;
    else      y = y - x;
}
z = x;
\%
% z = greatest-common-divisor
% of m and n
\end{verbatim}

partial correctness triple
\begin{align*}
\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) \}
\end{align*}

\textit{GCD} property
\begin{align*}
\gcd(X, Y, D) &:- X > Y, \quad X1 = X - Y, \quad \gcd(X1, Y, D). \\
\gcd(X, Y, D) &:- X < Y, \quad Y1 = Y - X, \quad \gcd(X, Y1, D). \\
\gcd(X, Y, D) &:- X = Y, \quad Y = D.
\end{align*}
CLP encoding of \textit{GCD}

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

\text{initConf}(\text{cf}(\text{cmd}(0, \text{asgn}(\text{int}(x), \text{int}(m)))))
\begin{align*}
&[[\text{int}(m), M], [\text{int}(n), N], [\text{int}(x), X], [\text{int}(y), Y], [\text{int}(z), Z])] : - \\
&M \geq 1, \quad N \geq 1. \quad \% \ \varphi_{init}(m,n)
\end{align*}

\text{errorConf}(\text{cf}(\text{cmd}(h, \text{halt})))
\begin{align*}
&[[\text{int}(m), M], [\text{int}(n), N], [\text{int}(x), X], [\text{int}(y), Y], [\text{int}(z), Z])] : - \\
&\text{gcd}(M, N, D), \quad D \neq Z. \quad \% \ \varphi_{error}(m,n,z)
\end{align*}

\textbf{Generation of VC's; Propagation of } \varphi_{error}(m,n,z) \textbf{ }

\textbf{Transformed } \textit{GCD}

incorrect :- M \geq 1, N \geq 1, M > N, X1 = M - N, Z \neq D, \text{ new1}(M, N, X1, N, Z, D).
incorrect :- M \geq 1, N \geq 1, M < N, Y1 = N - M, Z \neq D, \text{ new1}(M, N, M, Y1, Z, D).
new1(M, N, X, Y, Z, D) :- M \geq 1, N \geq 1, X > Y, X1 = X - Y, Z \neq D, \text{ new1}(M, N, X1, Y, Z).
new1(M, N, X, Y, Z, D) :- M \geq 1, N \geq 1, X < Y, Y1 = Y - X, Z \neq D, \text{ new1}(M, N, X, Y1, Z).

No constrained fact: the \textit{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.]