Program Verification using Constraint Handling Rules and Array Constraint Generalizations

Emanuele De Angelis[1,3], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1]University of Chieti-Pescara "G. d'Annunzio", Italy
[2]University of Rome "Tor Vergata", Italy
[3]CNR - Istituto di Analisi dei Sistemi ed Informatica, Rome, Italy

Workshop on:
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

Proving Partial Correctness of Imperative Programs

Consider a **program** and a **partial correctness triple**:

```
prog : while(x < n) {
           x = x+1 ;
           y = y+2 ;
       }
```

$$\{\, x=0 \land y=0 \land n\geq 1\}$$

prog

$$\{y > x\}$$

(A) Generate the **Verification Conditions** (VC's)

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

(B) If the VC's are **satisfiable** (i.e., there is an interpretation for $\mathrm{P}(x, y, n)$ 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*. $\mathtt{p(X,Y,N)}$:- $\mathtt{X=0,\ Y=0,\ N \geq 1}$. (a constrained fact)

2*. $\mathtt{p(X1,Y1,N)}$:- $\mathtt{X<N,\ X1=X+1,\ Y1=Y+2,\ p(X,Y,N)}$.

3*. $\mathtt{incorrect}$:- $\mathtt{X \geq N, Y \leq X, p(X,Y,N)}$.

Theorem: The VC's are **satisfiable** iff $\mathtt{incorrect} \notin$ **the least model** $M(V)$.

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

V': 4. $\mathtt{q(X1,Y1,N)}$:- $\mathtt{X<N, X>Y, Y \geq 0, X1=X+1, Y1=Y+2,\ q(X,Y,N)}$.

5. $\mathtt{incorrect}$:- $\mathtt{X \geq N, Y \leq X, Y \geq 0\ N \geq 1,\ q(X,Y,N)}$.

least model preserved: $\mathtt{incorrect} \notin M(V)$ iff $\mathtt{incorrect} \notin M(V')$

no constrained facts for q: $\mathtt{incorrect} \notin M(V')$

Thus, $\{x=0 \wedge y=0 \wedge n \geq 1\}\ \mathit{prog}\ \{y>x\}$ holds.

Encoding partial correctness of array programs into CLP

Encoding Partial Correctness into CLP

Consider the triple $\{\varphi_{init}\}$ *prog* $\{\neg\varphi_{error}\}$.

A program *prog* is incorrect w.r.t. φ_{init} and φ_{error}
if a final configuration satisfying φ_{error}
is reachable from an initial configuration satisfying φ_{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 `incorrect` $\in M(Int)$

A program *prog* is correct iff it is not incorrect.

tr(X,Y): the operational semantics

```
L: Id = Expr
```

```
tr( cf(cmd(L,asgn(Id,Expr)),S), cf(cmd(L1,C1),S1)) :-
     aeval(Expr,S,V),        evaluate expression
     update(Id,V,S,S1),      update store
     nextlabel(L,L1),        next label
     at(L1,C1).              next command
```

```
L: if(Expr) {
    L1: ...
  }
  else
    L2: ...
  }
```

```
tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-
     beval(Expr,S),          expression is true
     at(L1,C).               next command
tr( cf(cmd(L,ite(Expr,L1,L2)),S), cf(C,S)) :-
     beval(not(Expr),S),     expression is false
     at(L2,C).               next command
```

```
L: goto L1
```

```
tr( cf(cmd(L,goto(L1)),S), cf(C,S)) :-
     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: `S1`

transition:

```
tr( cf(cmd(L,asgn(elem(A,IE),E)),S),  old configuration cf
    cf(cmd(L1,C),S1) ) :-             new configuration cf

      eval(IE,S,I),                   evaluate index expr  IE
      eval(E,S,V),                    evaluate expression  E
      lookup(S,array(A),FA),          get array FA from store S
      write(FA,I,V,FA1),              update array FA, getting FA1
      update(S,array(A),FA1,S1),      update store S, getting S1

      nextlab(L,L1),                  next label L1
      at(L1,C).                       command C at next label
```

Running Example: Up Array Initialization

Program *UpInit*

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

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

$[2,?,?,?] \longrightarrow [2,3,?,?] \longrightarrow [2,3,4,?] \longrightarrow [2,3,4,5]$

Running Example: Up Array Initialization

Given the **program** *UpInit* and the **partial correctness triple**

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

$$\{i \geq 0 \;\wedge\; n \geq 1 \;\wedge\; n = dim(\mathtt{a})\}$$

UpInit

$$\{\forall j\; (0 \leq j \wedge j+1 < n \;\rightarrow\; \mathtt{a}[j] < \mathtt{a}[j+1])\}$$

CLP encoding of program *UpInit*

- A set of at(label, command) facts.
- while becomes ite + goto.
- a[i] becomes elem(a,i).

```
at(ℓ0,asgn(i,1)).
at(ℓ1,ite(less(i,n),ℓ2,ℓh)).
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(ℓh,halt).
```

CLP encoding of φ_{init} and φ_{error}

$$\mathtt{initConf}(\ell_0, \mathtt{I}, \mathtt{N}, \mathtt{A})\;\mathtt{:-}\;\; \mathtt{I} \geq 0,\; \mathtt{N} \geq 1.$$

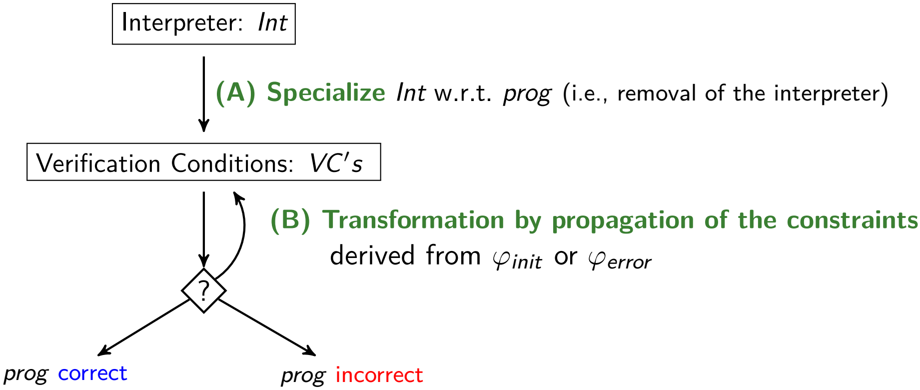
$$\mathtt{errorConf}(\ell_h, \mathtt{N}, \mathtt{A})\;\mathtt{:-}\;\; \mathtt{W} \geq 0,\; \mathtt{W}+1 < \mathtt{N},\; \boxed{\mathtt{Z} = \mathtt{W}+1,\; \mathtt{U} \geq \mathtt{V},}\; \mathtt{read}(\mathtt{A}, \boxed{\mathtt{W}, \mathtt{U}}), \mathtt{read}(\mathtt{A}, \boxed{\mathtt{Z}, \mathtt{V}}).$$

The array constraints: `read` and `write`

- if $\mathrm{a}[i] = v$, then `read(A,I,V)` holds
- if $\mathrm{a}[i] := v$, then `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



Interpreter: *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 φ_{init} or φ_{error}

prog correct

prog incorrect

- *prog* correct if no constrained facts appear in the $VC's$.
- *prog* incorrect if the fact `incorrect.` appears in the $VC's$.

Strategy for Specialiazion and Transformation

Transform

```
$TransfP = \emptyset$;
Defs = {incorrect :- errorConf(X), reach(X)};
while $\exists q \in$ Defs do
    Cls = Unfolding($q$);
    Cls = ConstraintReplacement(Cls);
    Cls = ClauseRemoval(Cls);
    Defs = (Defs − {$q$}) ∪ Definition$_{\text{array}}$(Cls);
    $TransfP$ = $TransfP$ ∪ Folding(Cls, Defs);
od
```

Generation of Verification Conditions

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,
             read(A,W,U), read(A,Z,V), new1(I,N,A).
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 $\mathcal{A} \models \forall\,(c_0 \leftrightarrow (c_1 \vee \ldots \vee c_n))$, where $\mathcal{A}$ is the Theory of Arrays

Then replace `H :- c0, d, G`
by `H :- c1, d, G,` ..., `H :- cn, d, G`

Constraint Handling Rules [Frühwirth et al.] for Constraint Replacement:

AC1. Array-Congruence-1: if i=j then a[i]=a[j]

$\mathtt{read(A, I, X)} \setminus \mathtt{read(A1, J, Y)} \Leftrightarrow \mathtt{A == A1, I = J} \mid \mathtt{X = Y}.$

AC2. Array-Congruence-2: if a[i]≠a[j] then i≠j

$\mathtt{read(A, I, X), read(A1, J, Y)} \Rightarrow \mathtt{A == A1, X <> Y} \mid \mathtt{I <> J}.$

ROW. Read-Over-Write: {a[i]=x; y=a[j]} if i=j then x=y

$\mathtt{write(A, I, X, A1)} \setminus \mathtt{read(A2, J, Y)} \Leftrightarrow \mathtt{A1 == A2} \mid$
$\mathtt{(I = J, X = Y)\ ;\ (I <> J, read(A,J,Y))}.$

Up Array Initialization

```
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:

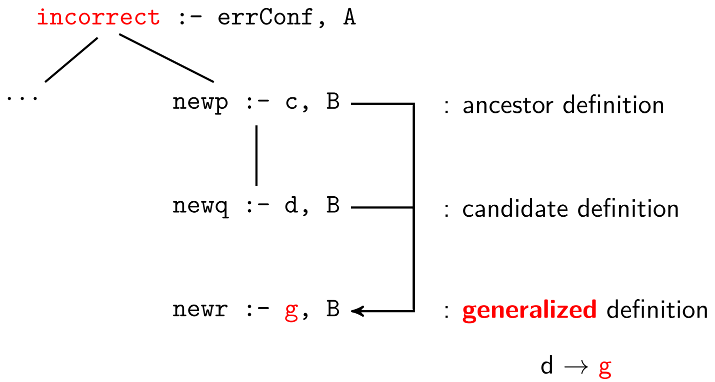
```
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., J=E, K=G,
  read(C,D,F), read(N,D,I), write(N,J,K,C), ~~read(C,E,G)~~,
write(N,J,K,C), read(C,E,G),
  reach(J,B,N).
new3(A,B,C) :- J=1+D, A=2+D, K=1+I, I<F, ..., J<>E,
  read(C,D,F), read(N,D,I), write(N,J,K,C), read(C,E,G),
write(N,J,K,C), read(C,E,G),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

Constraint Generalizations

Definitions are arranged as a tree:



`incorrect :- errConf, A`

...

`newp :- c, B` : ancestor definition

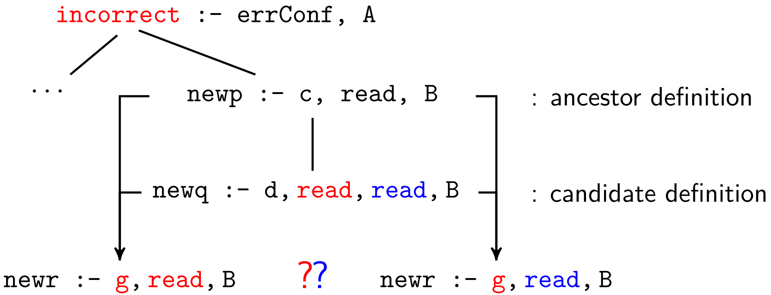
`newq :- d, B` : candidate definition

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

d → g

Generalization operators based on widening and convex-hull [Cousot-Cousot 77, Cousot-Halbwachs 78].

Array Constraint Generalizations



incorrect :- errConf, A

. . .

newp :- c, read, B : ancestor definition

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

newr :- g, read, B ?? newr :- g, read, B

- We decorate CLP variables with the associated identifiers of the imperative program.

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

incorrect :- Z = W + 1, W ≥ 0, W + 1 < N, U ≥ V, N ≤ I,
read(A, W^{j}, $U^{a[j]}$), read(A, Z^{j1}, $V^{a[j1]}$), new1(I, N, A).
new1(I1, N, B) :- 1 ≤ I, I < N, D = I − 1, I1 = I + 1, V = U + 1,
read(A, D^{i}, $U^{a[i]}$), write(A, I, V, B), new1(I, N, A).
new1(I, N, A) :- I = 1, N ≥ 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,`E^{j}`,`$G^{a[j]}$`), read(A,`F^{j1}`,`$H^{a[j1]}$`), 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,`E^{j}`,`$G^{a[j]}$`), read(A,`F^{j1}`,`$H^{a[j1]}$`), 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,`E^{j}`,`$G^{a[j]}$`), read(A,`F^{j1}`,`$H^{a[j1]}$`), reach(I,N,A).`

In the paper: a variable of the form G^{v} is encoded by `val(v,G)`.

Derived Verification Conditions

By applying the transformation strategy *Transform* to the verification conditions for *UpInit*, we get:

VC': Transformed verification conditions for *UpInit*

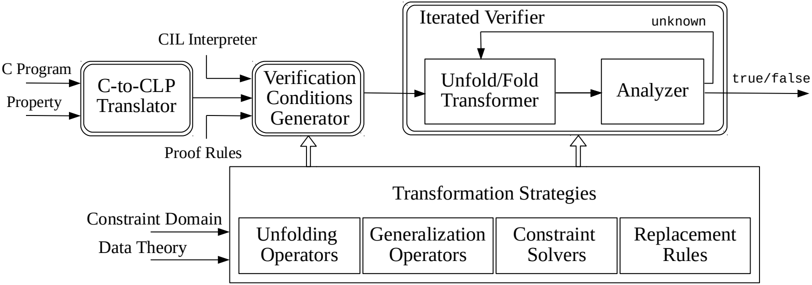
```
incorrect :- J1=J+1, J≥0, J1<I, AJ≥AJ1, D=I−1, N=I+1, Y=X+1,
     read(A, J, AJ), read(A, J1, AJ1), read(A, D, X), write(A, I, Y, B),
     new1(I, N, A).
new1(I1, N, B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, N≤I+2,
     I≥1, Z<I, Z≥1, N>I, U≥V, read(A, W, U), read(A, Z, V),
     read(A, D, X), write(A, I, Y, B), new5(I, N, A).
new5(I1, N, B) :- I1=I+1, Z=W+1, Y=X+1, D=I−1, I≥1,
     Z<I, Z≥1, N>I, U≥V, read(A, W, U), read(A, Z, V),
     read(A, D, X), write(A, I, Y, B), new5(I, N, A).
```

No constrained facts in VC': `incorrect` $\notin M(VC')$.
The program *UpInit* is correct.

Experimental results

VeriMAP

- The VeriMAP tool `http://map.uniroma2.it/VeriMAP`



Experimental evaluation

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

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

Proving recursively defined properties (1)

The *GCD* program

```
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
```

partial correctness triple

$$\varphi_{init}(m,n) \equiv \{\, m \geq 1 \land n \geq 1 \,\}$$

$$GCD$$

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

GCD property

```
gcd(X, Y, D) :- X > Y,  X1 = X - Y,  gcd(X1, Y, D).
gcd(X, Y, D) :- X < Y,  Y1 = Y - X,  gcd(X, Y1, D).
gcd(X, Y, D) :- X = Y,  Y = D.
```

Proving recursively defined properties (2)

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 $\varphi_{error}(m,n,z)$

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