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'
[2]University of Rome 'Tor Vergata'
[3]CNR - Istituto di Analisi dei Sistemi ed Informatica, Rome

CILC 2014
Torino, June 17, 2014

Proving Partial Correctness

Given the **program** *prog* and the **formal specification** φ

```
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** (VCs)

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 invariant
3.	$P(x, y, n) \land x \geq n \rightarrow y > x$	Exit

(B) prove they are **satisfiable**

If **satisfiable** then the correctness triple **hold**.

…Proving Partial Correctness

VCs are **satisfiable** if there is an **interpretation** that makes them true.

For instance, the interpretation

$$P(x, y, n) \equiv (x=0 \wedge y=0 \wedge n \geq 1) \vee y > x$$

makes the VCs true

$$\begin{array}{ll}
1'. & x=0 \wedge y=0 \wedge n \geq 1 \rightarrow (x=0 \wedge y=0 \wedge n \geq 1) \vee y > x \\
2'. & ((x=0 \wedge y=0 \wedge n \geq 1) \vee y > x) \wedge x < n \\
 & \quad \rightarrow (x+1=0 \wedge y+2=0 \wedge n \geq 1) \vee y+2 > x+1 \\
3'. & ((x=0 \wedge y=0 \wedge n \geq 1) \vee y > x) \wedge x \geq n \rightarrow y > x
\end{array}$$

and hence the triple $\{x=0 \wedge y=0 \wedge n \geq 1\}\ \mathit{prog}\ \{y > x\}$ **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 ≥ 1.                          Constrained fact
2. p(X1,Y1,N) :- X<N, X1=X+1, Y1=Y+2, p(X,Y,N).  Rule
4. incorrect :- X≥N, Y≤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 at.]
- 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(X1,Y1,N) :- $X<N$, $X1=X+1$, $Y1=Y+2$, p(X,Y,N).
4. incorrect :- $X \geq N$, $Y \leq X$, p(X, Y, N).

and derive the **equisatisfiable** V':

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

i.e., incorrect $\in M(V)$ *iff* incorrect $\in M(V')$.

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

Outline of the Talk

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

Encoding Partial Correctness into CLP

Given the specification $\{\varphi_{init}\}$ *prog* $\{\psi\}$ define $\varphi_{error} \equiv \neg\psi$

Definition (The interpreter *Int*)

`incorrect :- errorConf(X), reach(X).` | `X` satisfies φ_{error}

`reach(X) :- tr(X,Y), reach(Y).`

`reach(X) :- initConf(X).` | `X` satisfies φ_{init}

$+$ clauses for `tr` (the semantics of the programming language)

A program *prog* is **incorrect** w.r.t. φ_{init} and φ_{error}

if from an initial configuration satisfying φ_{init}

it is possible to reach a final configuration satisfying φ_{error}.

Otherwise, program *prog* is **correct**.

Theorem

prog is correct iff `incorrect` $\notin M(Int)$ *(the least model of Int)*

Running Example: Array Initialization

Given the **program** *SeqInit* and the **formal specification** φ

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

$$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$$
$$SeqInit$$
$$\{\forall j \ (0 \leq j \ \wedge \ j+1 < n \ \rightarrow \ a[j] < a[j+1])\}$$

CLP encoding of program SeqInit

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

$\mathtt{at}(\ell_0, \mathtt{asgn}(\mathtt{i}, 1))$.

$\mathtt{at}(\ell_1, \mathtt{ite}(\mathtt{less}(\mathtt{i}, \mathtt{n}), \ell_2, \ell_\mathtt{h}))$.

$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{elem}(\mathtt{a}, \mathtt{i}),$
$\quad \mathtt{plus}(\mathtt{elem}(\mathtt{a}, \mathtt{minus}(\mathtt{i}, 1)), 1)))$.

$\mathtt{at}(\ell_3, \mathtt{asgn}(\mathtt{i}, \mathtt{plus}(\mathtt{i}, 1)))$.

$\mathtt{at}(\ell_4, \mathtt{goto}(\ell_1))$.

$\mathtt{at}(\ell_\mathtt{h}, \mathtt{halt})$.

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

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

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

The Transformation-based Verification Method

Program transformation is a technique that

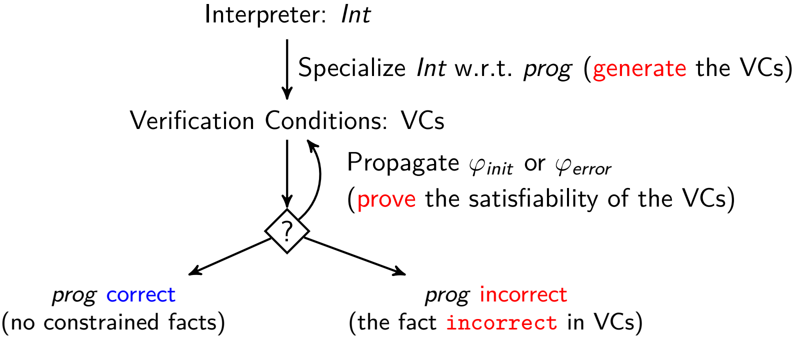
changes the syntax of a program

preserves its semantics.

Program Transformation of CLP can be used to

(A) generate the VCs

(B) prove the satisfiability of the VCs



Interpreter: *Int*

Specialize *Int* w.r.t. *prog* (generate the VCs)

Verification Conditions: VCs

Propagate φ_{init} or φ_{error}

(prove the satisfiability of the VCs)

?

prog correct

(no constrained facts)

prog incorrect

(the fact `incorrect` in VCs)

Rule-based Program Transformation

$$\begin{tikzcd}
\boxed{P} \arrow[d, "R"] \\
\mathrm{P}_1 \arrow[d, "R"] \\
\mathrm{P}_2 \arrow[d, "R"] \\
\vdots \arrow[d, "R"] \\
\boxed{\mathit{TransfP}}
\end{tikzcd}$$

Rule-based program transformation

- transformation **rules**:

  $R \in \{$ Definition, Unfolding, Folding, Clause Removal $\}$

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

  Theorem (Rules are semantics preserving)

  $\texttt{incorrect} \in M(P)$ *iff* $\texttt{incorrect} \in M(\mathit{TransfP})$

- 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
```

Verification Conditions generation

The specialization of *Int* w.r.t. *prog* removes all references to:

- `tr` (i.e., the operational semantics of the imperative language)
- `at` (i.e., the encoding of *prog*)

The Specialized Interpreter for SeqInit (Verification Conditions)

```
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.

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 $- \{q\}) \cup$ $\text{Define}_{\text{array}}$(Cls);
    $TransfP = TransfP \cup$ Fold(Cls, Defs);
od
```

Constraint Replacement Rule

If $\mathcal{A} \models \forall(\mathtt{c}_0 \leftrightarrow (\mathtt{c}_1 \vee \ldots \vee \mathtt{c}_n))$, where $\mathcal{A}$ is the Theory of Arrays

Then replace $\mathtt{H :- c_0, d, G}$

by $\mathtt{H :- c_1, d, G, \ldots, H :- c_n, d, G}$

Constraint Handling Rules for Constraint Replacement:

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

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

CAC Contrapositive Array Congruence (if a[i]≠a[j] then i≠j)

$\mathtt{read(A1, I, X), read(A2, J, Y) \Rightarrow A1 == A2, X <> Y \mid I <> J.}$

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

$\mathtt{write(A1, I, X, A2) \setminus read(A3, J, Y) \Leftrightarrow A2 == A3 \mid}$

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

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 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≤-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

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, . . . , J=E, K=G,
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≤-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

```
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, ..., J=E, K=G,
  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,  ..., J<>E,
  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

```
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, ..., J=E, K=G,
  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,  ..., J<>E,
  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).
```

Definition introduction

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:

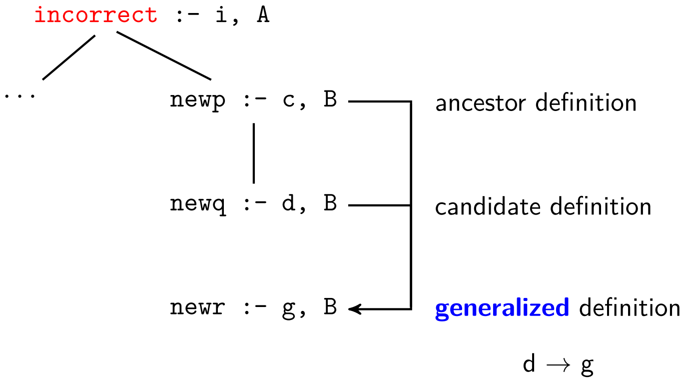
- to ensure the termination of the transformation,
- to generate program invariants,

... two somewhat conflicting requirements:

- efficiency, to introduce as few definitions as possible,
- precision, to prove as many properties as possible.

Constraint Generalizations

Definitions are arranged as a tree:



`incorrect :- i, A`

`...`

`newp :- c, B` ancestor definition

`newq :- d, B` candidate definition

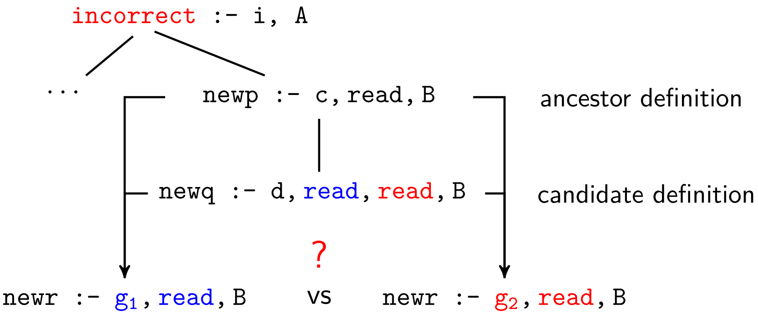
`newr :- g, B` **generalized** definition

d → g

Generalization operators based on widening and convex-hull.

Array Constraint Generalizations

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



`incorrect :- i, A`

...

`newp :- c, read, B`

ancestor definition

`newq :- d, read, read, B`

candidate definition

?

`newr :-` g_1`, read, B` vs `newr :-` g_2`, read, B`

The Specialized Interpreter for SeqInit (Verification Conditions)

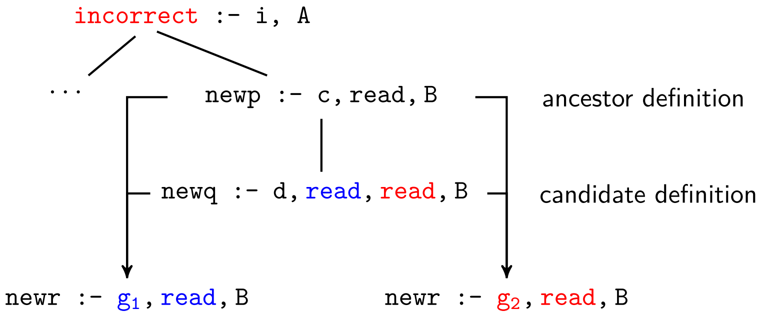
incorrect :- $Z=W+1,\ W\geq 0,\ W+1<N,\ U\geq V,\ N\leq I,$
$\quad$ $\mathtt{read}(A, W^{j}, U^{a[j]}),\ \mathtt{read}(A, Z^{j1}, V^{a[j1]}),\ \mathtt{new1}(I,N,A).$

new1(I1,N,B) :- $1\leq I,\ I<N,\ D=I-1,\ I1=I+1,\ V=U+1,$
$\quad$ $\mathtt{read}(A, D^{i}, U^{a[i]}),\ \mathtt{write}(A,I,V,B),\ \mathtt{new1}(I,N,A).$

new1(I,N,A) :- $I=1,\ N\geq 1.$

Array Constraint Generalizations

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



incorrect :- i, A

...

newp :- c, read, B

ancestor definition

newq :- d, read, read, B

candidate definition

newr :- g_1, read, B

newr :- g_2, read, B

The Specialized Interpreter for SeqInit (Verification Conditions)

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

new1$(I1, N, B)$:- $1\leq I,\ I<N,\ D=I-1,\ I1=I+1,\ V=U+1,$
read$(A, D^{i}, U^{a[i]})$, write(A, I, V, B), new1(I, N, A).

new1(I, N, A) :- $I=1,\ N\geq 1$.

Array Initialization

- ancestor definition:

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

- candidate definition:

$\mathtt{new4(I,N,A)}$:- $E+1=F, E\geq 0, I>F, G\geq H, I=1+I1, I1+2\leq C, N\leq I1+3,$
$\mathtt{read}(A,E^{j},G^{a[j]}), \mathtt{read}(A,F^{j1},H^{a[j1]}), \mathtt{read}(A,P^{i},Q^{a[i]}),$
$\mathtt{reach}(I,N,A).$

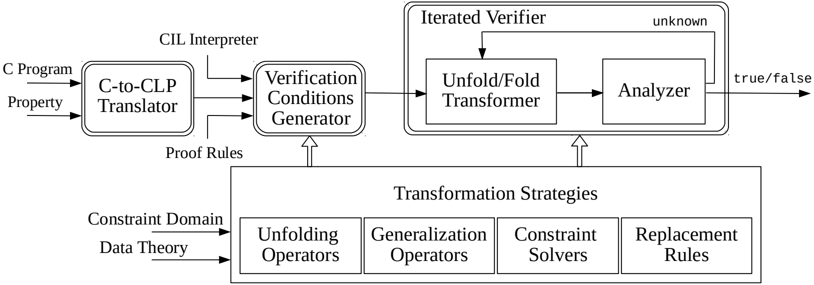
- generalized definition:

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

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

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
precision	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

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