Verification of Imperative Programs through Transformation of Constraint Logic Programs

Emanuele De Angelis[1], 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

LSV-ENS Cachan, Feb 12, 2014

Proving Partial Correctness of Imperative Programs

Given the *program* $prog$:

$x = 0;\ \ y = 0;$
$\texttt{while}\ (x < n)\ \{x = x + 1;\ \ y = y + 2\}$

and the *specification*:

$\{n \geq 1\}\ prog\ \{y > x\}$

Proving Partial Correctness of Imperative Programs

Given the *program* *prog*:

$x = 0;\;\; y = 0;$
$\texttt{while}\ (x < n)\ \{x = x + 1;\;\; y = y + 2\}$

and the *specification*:

$\{n \geq 1\}\ prog\ \{y > x\}$

Generate the *verification conditions* (VCs):

1. $x = 0 \wedge y = 0 \wedge n \geq 1 \;\rightarrow\; P(x, y, n)$ Initialization
2. $P(x, y, n) \wedge x < n \;\rightarrow\; P(x + 1, y + 2, n)$ Loop invariant
3. $P(x, y, n) \wedge x \geq n \;\rightarrow\; y > x$ Exit

and prove they are *satisfiable*, i.e., we can find an interpretation for P that makes the VCs true.

Proving Satisfiability of Verification Conditions

The *interpretation*

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

makes the *VCs true*

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

and hence the specification $\{n \geq 1\}$ *prog* $\{y > x\}$ is *valid*.

Proving Satisfiability of Verification Conditions

The *interpretation*

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

makes the *VCs true*

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

and hence the specification $\{n \geq 1\}\, prog\, \{y > x\}$ is *valid*.

Problem: How to find the interpretation for P automatically?

Methods Based on Horn Clauses with Constraints (CLP)

- The VCs are a set of *Horn clauses with constraints*

Methods Based on Horn Clauses with Constraints (CLP)

- The VCs are a set of *Horn clauses with constraints*
- or, equivalently, a *constraint logic program*:

1. $x=0 \wedge y=0 \wedge n\geq 1 \rightarrow P(x,y,n)$
2. $P(x,y,n) \wedge x<n \rightarrow P(x+1,y+2,n)$
4. $P(x,y,n) \wedge x\geq n \wedge y\leq x \rightarrow \mathit{false}$

VCs *satisfiable* iff *false* *not* in the *least model* of V.

Methods Based on Horn Clauses with Constraints (CLP)

- The VCs are a set of *Horn clauses with constraints*
- or, equivalently, a *constraint logic program*:

  1. $x\!=\!0 \wedge y\!=\!0 \wedge n\!\geq\!1 \rightarrow P(x,y,n)$
  2. $P(x,y,n) \wedge x\!<\!n \rightarrow P(x+1,y+2,n)$
  4. $P(x,y,n) \wedge x\!\geq\!n \wedge y\!\leq\!x \rightarrow \mathit{false}$

  VCs *satisfiable* iff *false* *not* in the *least model* of V.
- Methods for proving the satisfiability of VCs in the framework of CHC/CLP:
  - CounterExample Guided Abstraction Refinement, Interpolation, Satisfiability Modulo Theories [McMillan, Rybalchenko, Björner, Poppea et al.]
  - Symbolic execution of CLP [Jaffar, Navas, Santosa et al.]
  - Static Analysis and *Transformation of CLP* [Gallagher, Albert, DFPP et al.]

A Transformation-based Method

- Apply to V transformations that *preserve the least model*:

  1. $x = 0 \wedge y = 0 \wedge n \geq 1 \rightarrow P(x, y, n)$ *Constrained fact*
  2. $P(x, y, n) \wedge x < n \rightarrow P(x+1, y+2, n)$
  4. $P(x, y, n) \wedge x \geq n \wedge y \leq x \rightarrow \mathit{false}$

  and derive the *equisatisfiable* V':

  5. $Q(x, y, n) \wedge x < n \wedge x > y \wedge y \geq 0 \rightarrow Q(x+1, y+2, n)$
  6. $Q(x, y, n) \wedge x \geq n \wedge x \geq y \wedge y \geq 0 \wedge n \geq 1 \rightarrow \mathit{false}$

A Transformation-based Method

- Apply to V transformations that *preserve the least model*:

1. $x = 0 \wedge y = 0 \wedge n \geq 1 \rightarrow P(x, y, n)$ *Constrained fact*
2. $P(x, y, n) \wedge x < n \rightarrow P(x + 1, y + 2, n)$
4. $P(x, y, n) \wedge x \geq n \wedge y \leq x \rightarrow \mathit{false}$

and derive the *equisatisfiable* V':

5. $Q(x, y, n) \wedge x < n \wedge x > y \wedge y \geq 0 \rightarrow Q(x + 1, y + 2, n)$
6. $Q(x, y, n) \wedge x \geq n \wedge x \geq y \wedge y \geq 0 \wedge n \geq 1 \rightarrow \mathit{false}$

No constrained facts: V' satisfiable with $Q(x, y, n) \equiv \mathit{false}$.

A Transformation-based Method

- Apply to V transformations that *preserve the least model*:

  1. $x = 0 \wedge y = 0 \wedge n \geq 1 \rightarrow P(x, y, n)$ *Constrained fact*
  2. $P(x, y, n) \wedge x < n \rightarrow P(x + 1, y + 2, n)$
  4. $P(x, y, n) \wedge x \geq n \wedge y \leq x \rightarrow \mathit{false}$

  and derive the *equisatisfiable* V':

  5. $Q(x, y, n) \wedge x < n \wedge x > y \wedge y \geq 0 \rightarrow Q(x + 1, y + 2, n)$
  6. $Q(x, y, n) \wedge x \geq n \wedge x \geq y \wedge y \geq 0 \wedge n \geq 1 \rightarrow \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x, y, n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?

A Transformation-based Method

- Apply to V transformations that *preserve the least model*:

  1. $x = 0 \land y = 0 \land n \geq 1 \;\rightarrow\; P(x, y, n)$ *Constrained fact*
  2. $P(x, y, n) \land x < n \;\rightarrow\; P(x + 1, y + 2, n)$
  4. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow\; \mathit{false}$

  and derive the *equisatisfiable* V':

  5. $Q(x, y, n) \land x < n \land x > y \land y \geq 0 \;\rightarrow\; Q(x + 1, y + 2, n)$
  6. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \;\rightarrow\; \mathit{false}$

  No constrained facts: V' satisfiable with $Q(x, y, n) \equiv \mathit{false}$.

- Problem: How to transform V into V' automatically?

- Some transformation strategies for programs over integers [PEPM-13] and arrays [VMCAI-14].

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving *transformation rules and strategies*
  - *VC generation* by specialization of the interpreter
  - *VC transformation* by propagation of the property to be verified

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving *transformation rules and strategies*
  - *VC generation* by specialization of the interpreter
  - *VC transformation* by propagation of the property to be verified
- Improving precision via *iterated* VC transformation

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving *transformation rules and strategies*
  - *VC generation* by specialization of the interpreter
  - *VC transformation* by propagation of the property to be verified
- Improving precision via *iterated* VC transformation
- Verifying array programs via *constraint replacement*

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving *transformation rules and strategies*
  - *VC generation* by specialization of the interpreter
  - *VC transformation* by propagation of the property to be verified
- Improving precision via *iterated* VC transformation
- Verifying array programs via *constraint replacement*
- *Recursively* defined properties

Outline of the Talk

- Constraint Logic Programming as a *metalanguage* for representing
  - the imperative program
  - the semantics of the imperative language (*interpreter*)
  - the property to be verified
- Verification method based on CLP program transformation
  - Semantics-preserving *transformation rules and strategies*
  - *VC generation* by specialization of the interpreter
  - *VC transformation* by propagation of the property to be verified
- Improving precision via *iterated* VC transformation
- Verifying array programs via *constraint replacement*
- *Recursively* defined properties
- Experimental evaluation: The VeriMAP system

CLP with integer constraints

- A CLP *clause* is an implication $\mathtt{c} \land \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where `H` is an atom, `c` is a constraint, and `G` is a conjunction of atoms

CLP with integer constraints

- A CLP *clause* is an implication $\mathtt{c} \wedge \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where `H` is an atom, `c` is a constraint, and `G` is a conjunction of atoms

- A *constraint* is a conjunction of linear equalities/inequalities over integers ($\mathrm{p}_1 = \mathrm{p}_2$, $\mathrm{p}_1 \geq \mathrm{p}_2$, $\mathrm{p}_1 > \mathrm{p}_2$)

CLP with integer constraints

- A CLP *clause* is an implication $\texttt{c} \wedge \texttt{G} \rightarrow \texttt{H}$, written as:

  ```
  H :- c, G.
  ```

  where H is an atom, c is a constraint, and G is a conjunction of atoms

- A *constraint* is a conjunction of linear equalities/inequalities over integers ($p_1 = p_2$, $p_1 \geq p_2$, $p_1 > p_2$)

- A CLP *program* is a set of CLP clauses

CLP with integer constraints

- A CLP *clause* is an implication $\mathtt{c} \wedge \mathtt{G} \rightarrow \mathtt{H}$, written as:

  ```
  H :- c, G.
  ```

  where H is an atom, c is a constraint, and G is a conjunction of atoms

- A *constraint* is a conjunction of linear equalities/inequalities over integers ($p_1 = p_2$, $p_1 \geq p_2$, $p_1 > p_2$)

- A CLP *program* is a set of CLP clauses

- Semantics: *least model* of the program with the fixed interpretation of constraints.

Imperative Programs over Integers

- We consider an imperative language with integer variables, assignment, if-else, while-loop, and goto.
- Program *increase*:

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

- Partial Correctness Specification

$$\{x = 0 \land y = 0\}\ \textit{increase}\ \{x \leq y\}$$

Encoding of an Imperative Program into CLP

A program is represented as a set of atoms $\mathtt{at}(\mathit{label}, \mathit{command})$.

Program *increase*:

ℓ_0 : `while(x < n){`
ℓ_1 : `x=x+1;`
ℓ_2 : `y=x+y;`
ℓ_3 : `}`

CLP encoding of *increase*:

$\mathtt{at}(\ell_0, \mathtt{ite}(\mathtt{less}(\mathtt{int}(\mathtt{x}), \mathtt{int}(\mathtt{n})), \ell_1, \ell_h)).$
$\mathtt{at}(\ell_1, \mathtt{asgn}(\mathtt{int}(\mathtt{x}), \mathtt{plus}(\mathtt{int}(\mathtt{x}), \mathtt{int}(1)))).$
$\mathtt{at}(\ell_2, \mathtt{asgn}(\mathtt{int}(\mathtt{y}), \mathtt{plus}(\mathtt{int}(\mathtt{x}), \mathtt{int}(\mathtt{y})))).$
$\mathtt{at}(\ell_3, \mathtt{goto}(\ell_0)).$
$\mathtt{at}(\ell_h, \mathtt{halt}).$

CLP encoding of the operational semantics (1)

A *transition semantics* is defined by:

- a set of *configurations*, i.e., a CLP term: $\mathtt{cf}(\mathtt{C},\mathtt{S})$

  where:

  - C is a labeled *command*
  - S is a *store*,
    i.e., a list of [variable identifier, value] pairs:

    $$[[\mathtt{int}(\mathtt{x}),2],\ [\mathtt{int}(\mathtt{y}),3]]$$

- a *transition relation*: $\mathtt{tr}(\mathtt{cf}(\mathtt{C},\mathtt{S}),\ \mathtt{cf}(\mathtt{C1},\mathtt{S1}))$

`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) {`
`goto L1:`
`} else`
`goto 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).` `at(L1,C).` *next command*

CLP encoding of (in)correctness

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

Definition (Program Incorrectness)

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*.

CLP encoding of (in)correctness

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

Definition (Program Incorrectness)

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*.

Definition (CLP encoding of incorrectness: The interpreter *Int*)

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

`initConf(X)` $\equiv$ `X` is a configuration satisfying φ_{init}

`errorConf(X)` $\equiv$ `X` is a configuration satisfying φ_{error}

CLP encoding of (in)correctness

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

Definition (Program Incorrectness)

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*.

Definition (CLP encoding of incorrectness: The interpreter *Int*)

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

`initConf(X)` $\equiv$ `X` is a configuration satisfying φ_{init}

`errorConf(X)` $\equiv$ `X` is a configuration satisfying φ_{error}

Theorem (Correctness of Encoding)

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

Partial Correctness Specification

$\{x = 0 \land y = 0\}$ φ_{init}

increase

$\{x \leq y\}$ ψ

$\{x > y\}$ $\varphi_{error} \equiv \neg\psi$

Partial Correctness Specification

$\{x = 0 \wedge y = 0\}$ φ_{init}

increase

$\{x \leq y\}$ ψ

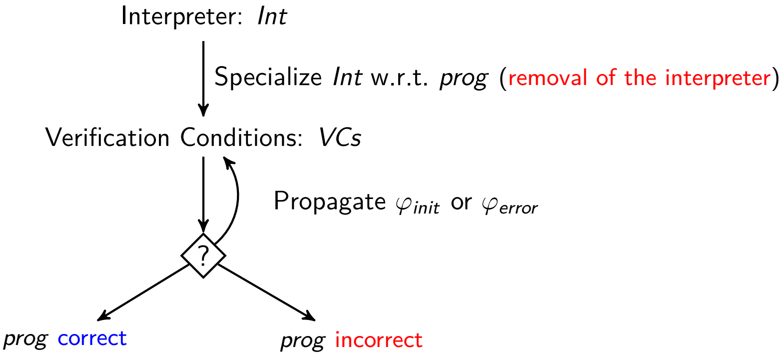
$\{x > y\}$ $\varphi_{error} \equiv \neg\psi$

Initial and Error Configurations

```
initConf(cf(cmd(0,ite(...)), [[int(x),X],[int(y),Y],[int(n),N]]))
          :- X=0, Y=0.
errorConf(cf(cmd(h,halt), [[int(x),X],[int(y),Y],[int(n),N]]))
          :- X>Y.
```

φ_{init} φ_{error}

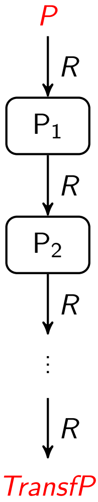
The Transformation-based Verification Method



- *prog* correct if no constrained facts appear in the VCs.
- *prog* incorrect if the fact `incorrect.` appears in the VCs.

Unfold/Fold Program Transformation

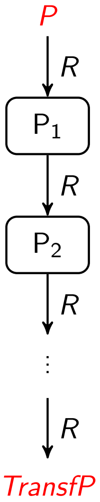
[Burstall-Darlington 77,Tamaki-Sato 84, Etalle-Gabbrielli 96]



- transformation *rules*:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal$\}$

Unfold/Fold Program Transformation

[Burstall-Darlington 77,Tamaki-Sato 84, Etalle-Gabbrielli 96]

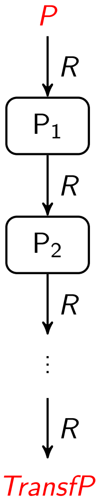


- transformation *rules*:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal$\}$
- the transformation rules *preserve the least model*:

$$\boxed{\texttt{incorrect} \in M(\mathrm{P}) \quad \textit{iff} \quad \texttt{incorrect} \in M(\mathrm{TransfP})}$$

Unfold/Fold Program Transformation

[Burstall-Darlington 77,Tamaki-Sato 84, Etalle-Gabbrielli 96]



- transformation *rules*:
  $R \in \{$ Definition,
  Unfolding,
  Folding,
  Clause Removal$\}$

- the transformation rules *preserve the least model*:

$$\boxed{\texttt{incorrect} \in M(\mathrm{P}) \quad \textit{iff} \quad \texttt{incorrect} \in M(\mathrm{TransfP})}$$

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

Rules for Transforming CLP Programs

R1. Definition. Introducing a new predicate (e.g., a loop invariant)

```
newp(X) :- c, A
```

Rules for Transforming CLP Programs

R1. Definition. Introducing a new predicate (e.g., a loop invariant)

$\texttt{newp(X) :- c, A}$

R2. Unfolding. A symbolic evaluation step (resolution)

given $\quad \texttt{H :- c, }\underline{\texttt{A}}\texttt{, G}$

$\qquad \underline{\texttt{A}}\texttt{ :- }\texttt{d}_1, \texttt{G}_1, \quad \ldots, \quad \underline{\texttt{A}}\texttt{ :- }\texttt{d}_\texttt{m}, \texttt{G}_\texttt{m}$

derive $\quad \texttt{H :- c, }\texttt{d}_1, \texttt{G}_1\texttt{, G}, \quad \ldots, \quad \texttt{H :- c, }\texttt{d}_\texttt{m}, \texttt{G}_\texttt{m}\texttt{, G}$

Rules for Transforming CLP Programs

R1. Definition. Introducing a new predicate (e.g., a loop invariant)

$\mathtt{newp(X)\ {:}{-}\ c, A}$

R2. Unfolding. A symbolic evaluation step (resolution)

given $\quad \mathtt{H\ {:}{-}\ c,\ \underline{A},\ G}$

$\quad \mathtt{\underline{A}\ {:}{-}\ d_1, G_1,\ \ \ldots,\ \ \underline{A}\ {:}{-}\ d_m, G_m}$

derive $\quad \mathtt{H\ {:}{-}\ c, d_1, G_1, G,\ \ \ldots,\ \ H\ {:}{-}\ c, d_m, G_m, G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

given $\quad \mathtt{H\ {:}{-}\ d, \underline{A}, G}$

$\quad \mathtt{newp(X)\ {:}{-}\ c, \underline{A}} \qquad$ and $\qquad \mathtt{d} \rightarrow \mathtt{c}$

derive $\quad \mathtt{H\ {:}{-}\ d, newp(X), G}$

Rules for Transforming CLP Programs

R1. Definition. Introducing a new predicate (e.g., a loop invariant)

$$\mathtt{newp(X)\ :\!-\ c, A}$$

R2. Unfolding. A symbolic evaluation step (resolution)

given $\quad \mathtt{H\ :\!-\ c,\ \underline{A},\ G}$

$\quad\quad\quad \mathtt{\underline{A}\ :\!-\ d_1, G_1,\ \ \ldots,\ \ \underline{A}\ :\!-\ d_m, G_m}$

derive $\quad \mathtt{H\ :\!-\ c, d_1, G_1, G,\ \ \ldots,\ \ H\ :\!-\ c, d_m, G_m, G}$

R3. Folding. Matching a predicate definition (e.g., a loop invariant)

given $\quad \mathtt{H\ :\!-\ d, \underline{A}, G}$

$\quad\quad\quad \mathtt{newp(X)\ :\!-\ c, \underline{A}}$ $\quad$ and $\quad \mathtt{d} \rightarrow \mathtt{c}$

derive $\quad \mathtt{H\ :\!-\ d, newp(X), G}$

R4. Clause Removal. Removal of clauses with unsatisfiable constraint or subsumed by others

The Transformation Strategy

Transform(*P*)

TransfP $= \emptyset$;
Defs $= \{$`incorrect :- initConf(X), reach(X)`$\}$;
while $\exists cl \in$ Defs **do**
  Cls $=$ Unfold(cl);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $- \{cl\}) \cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

The Transformation Strategy

Transform(*P*)

TransfP $= \emptyset$;
Defs $= \{$`incorrect :- initConf(X), reach(X)`$\}$;
while $\exists cl \in$ Defs **do**
  Cls $=$ Unfold(cl);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $-$ $\{cl\}$) $\cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

Theorem (Termination and Correctness of the Transformation Strategy)

- *Transform*(*P*) *terminates for all P;*
- `incorrect` $\in M$(P) *iff* `incorrect` $\in M$(TransfP)

Generalization Strategies

- The most critical transformation step during the unfold/fold transformation strategy is the *introduction of new predicate definitions* to be used for folding.

Generalization Strategies

- The most critical transformation step during the unfold/fold transformation strategy is the *introduction of new predicate definitions* to be used for folding.
- Given `p(X) :- c(X,Y), q(Y).`

  Introduce `newp(Y) :- d(Y), q(Y).`

  where $c(X,Y) \to d(Y)$ (`d(Y)` is a *generalization* of `c(X,Y)`)

  and *fold*: `p(X) :- c(X,Y), newp(Y).`

Generalization Strategies

- The most critical transformation step during the unfold/fold transformation strategy is the *introduction of new predicate definitions* to be used for folding.

- Given `p(X) :- c(X,Y), q(Y).`

  Introduce `newp(Y) :- d(Y), q(Y).`

  where $\mathtt{c}(\mathtt{X},\mathtt{Y}) \rightarrow \mathtt{d}(\mathtt{Y})$ (`d(Y)` is a *generalization* of `c(X,Y)`)

  and *fold*: `p(X) :- c(X,Y), newp(Y).`

- Generalization strategies based on *widening* and *convex-hull* of linear constraints.

Generating Verification Conditions via Specialization

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*)

Generating Verification Conditions via Specialization

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 *increase* (Verification Conditions)

```
incorrect :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

Generating Verification Conditions via Specialization

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 *increase* (Verification Conditions)

```
incorrect :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

- New predicates correspond to a subset of the *program points*:

```
new1(X,Y,N) :- reach(cf(cmd(0,ite(...)),
                  [[int(x),X],[int(y),Y],[int(n),N]])).
```

Generating Verification Conditions via Specialization

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 *increase* (Verification Conditions)

```
incorrect :- X=0, Y=0, new1(X,Y,N).
new1(X,Y,N) :- X<N, X1=X+1, Y1=X1+Y, new1(X1,Y1,N).
new1(X,Y,N) :- X≥N, X>Y.
```

- New predicates correspond to a subset of the *program points*:
  ```
  new1(X,Y,N) :- reach(cf(cmd(0,ite(...)),
                   [[int(x),X],[int(y),Y],[int(n),N]])).
  ```

- The fact `incorrect.` is not in VCs: we cannot infer that *increase* is incorrect.
  A constrained fact is in VCs: we cannot infer that *increase* is correct.

Propagation of φ_{init}

The verification conditions VCs are specialized w.r.t. the initial configuration.

Specialized Verification Conditions for *increase*

…propagating the constraint $\mathtt{X}=0,\ \mathtt{Y}=0$.

$\mathtt{incorrect}$:- $\mathtt{N}>0,\ \mathtt{X1}=1,\ \mathtt{Y1}=1,\ \mathtt{new2(X1,Y1,N)}.$
$\mathtt{new2(X,Y,N)}$:- $\mathtt{X}=1,\ \mathtt{Y}=1,\ \mathtt{N}>1,\ \mathtt{X1}=2,\ \mathtt{Y1}=3,\ \mathtt{new3(X1,Y1,N)}.$
$\mathtt{new3(X,Y,N)}$:- $\mathtt{X1}\geq 1,\ \mathtt{Y1}\geq \mathtt{X1},\ \mathtt{X}<\mathtt{N},\mathtt{X1}=\mathtt{X}+1,\ \mathtt{Y1}=\mathtt{X1}+\mathtt{Y},\ \mathtt{new3(X1,Y1,N)}.$
$\mathtt{new3(X,Y,N)}$:- $\mathtt{Y}\geq 1,\ \mathtt{N}>0,\ \mathtt{X}\geq \mathtt{N},\ \mathtt{X}>\mathtt{Y}.$

The fact `incorrect.` is not in VCs: we cannot infer that *increase* is incorrect.
A constrained fact is in VCs: we cannot infer that *increase* is correct.

Introduction of new definitions by generalization

1. `incorrect :- X=0, Y=0, new1(X,Y,N).`
2. `new2(X,Y,N) :- X=1, Y=1, N>0, new1(X,Y,N).`

Candidate new definition:

`new3(Xr,Yr,Nr) :- Xr=1, Yr=1, X=2, Y=3, N>1, new1(X,Y,N).`

The transformation strategy might introduce infinitely many new definitions. Generalization is needed.

Generalization (based on widening):

3. `new3(X,Y,N) :-` $X \geq 1$, $Y \geq 1$, `N>0, new1(X,Y,N).`

Program Reversal

P:

```
incorrect :- a(X), p(X).
p(X) :- c(X,Y), p(Y).
p(X) :- b(X).
```

↓ Reversal

RevP:

```
incorrect :- b(X), p(X).
p(Y) :- c(X,Y), p(X).
p(X) :- a(X).
```

incorrect $\in M(\text{P})$ iff incorrect $\in M(\text{RevP})$

Propagation of φ_{error}

Specialized Verification Conditions for *increase*

$\mathtt{incorrect} \;\text{:-}\; N>0,\ X1=1,\ Y1=1,\ new2(X1,Y1,N).$

$\underline{new2(X,Y,N)} \;\text{:-}\; X=1,\ Y=1,\ N>1,\ X1=2,\ Y1=3,\ \underline{new3(X1,Y1,N)}.$

$\underline{new3(X,Y,N)} \;\text{:-}\; X1\geq 1,\ Y1\geq X1,\ X<N, X1=X+1,\ Y1=X1+Y, \underline{new3(X1,Y1,N)}.$

$new3(X,Y,N) \;\text{:-}\; Y\geq 1,\ N>0,\ X\geq N,\ X>Y.$

Reversed VCs

$\mathtt{incorrect} \;\text{:-}\; Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ new3(X,Y,N).$

$\underline{new3(X1,Y1,N)} \;\text{:-}\; X1\geq 1,\ Y1\geq X1,\ X<N, X1=X+1,\ Y1=X1+Y,\ \underline{new3(X,Y,N)}.$

$\underline{new3(X1,Y1,N)} \;\text{:-}\; X=1,\ Y=1,\ N>1,\ X1=2,\ Y1=3,\ \underline{new2(X,Y,N)}.$

$new2(X1,Y1,N) \;\text{:-}\; N>0,\ X1=1,\ Y1=1.$

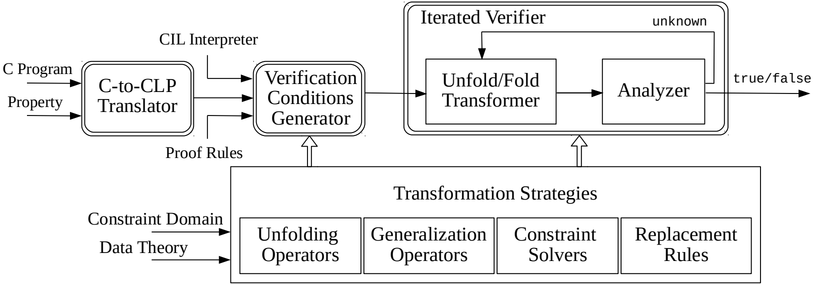
Specialized VCs

by propagating the constraint $Y\geq 1,\ N>0,\ X\geq N,\ X>Y.$

$\mathtt{incorrect} \;\text{:-}\; Y\geq 1,\ N>0,\ X\geq N,\ X>Y,\ new4(X,Y,N).$

No constrained facts: *increase* is correct.

The VeriMAP tool `http://map.uniroma2.it/VeriMAP`
[DFPP PEPM 2013, VMCAI 2014, TACAS 2014]



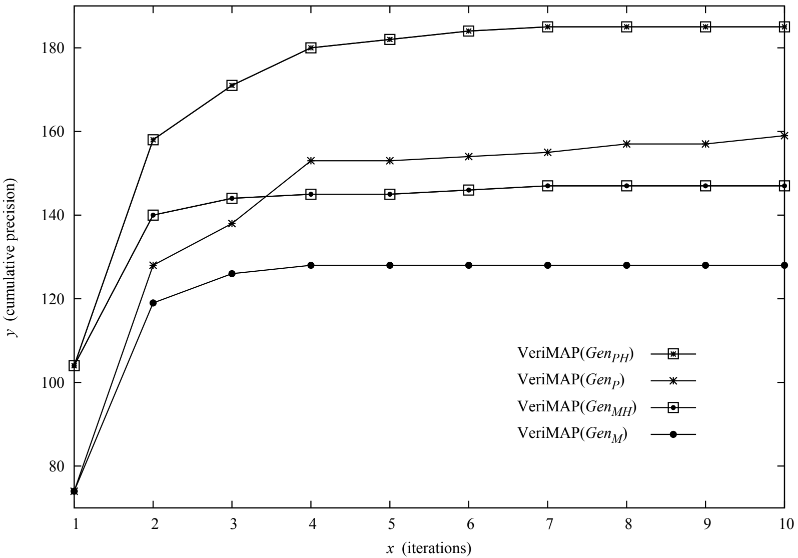
Experimental Evaluation

216 examples taken from: DAGGER, TRACER, InvGen, and TACAS 2013 Software Verification Competition.

		VeriMAP	ARMC	HSF(C)	TRACER
1	*correct answers*	185	138	160	103
2	safe problems	154	112	138	85
3	unsafe problems	31	26	22	18
4	*incorrect answers*	0	9	4	14
5	false alarms	0	8	3	14
6	missed bugs	0	1	1	0
7	*errors*	0	18	0	22
8	*timed-out problems*	31	51	52	77
9	*total score*	339 (0)	210 (-40)	278 (-20)	132 (-56)
10	*total time*	10717.34	15788.21	15770.33	23259.19
11	*average time*	57.93	114.41	98.56	225.82

- ARMC [Podelski, Rybalchenko PADL 2007]
- HSF(C) [Grebenshchikov et al. TACAS 2012]
- TRACER [Jaffar, Murali, Navas, Santosa CAV 2012]

Improving Precision by Iteration



Verifying Array Programs

An Example: Array Initialization.

Program *SeqInit*

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

Verifying Array Programs

An Example: Array Initialization.

Program *SeqInit*

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

An Execution

$[4, \_, \_, \_] \Longrightarrow [4, 5, \_, \_] \Longrightarrow [4, 5, 6, \_] \Longrightarrow [4, 5, 6, 7]$

Verifying Array Programs

An Example: Array Initialization.

Program *SeqInit*

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

An Execution

$[4, \_, \_, \_] \Longrightarrow [4, 5, \_, \_] \Longrightarrow [4, 5, 6, \_] \Longrightarrow [4, 5, 6, 7]$

Partial Correctness Specification

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

CLP encoding of the operational semantics (3)

The transition for *array assignment*

- *command*: $\mathtt{L} : \mathtt{a}[\mathtt{ie}] = \mathtt{e}$
- *store*: $\mathtt{S}$
- *transition*:

```
tr(cf(cmd(L,asgn(elem(A,IE),E)),S),      source configuration
   cf(cmd(L1,C),S1)) :-                  target configuration
     eval(IE,S,I),                       evaluate index expr
     eval(E,S,V),                        evaluate expression
     lookup(S,array(A),FA),              get array from store
     write(FA,I,V,FA1),                  update array
     update(S,array(A),FA1,S1),          update store
     nextlab(L,L1),                      next label
     at(L1,C).                           next command
```

Partial Correctness Specification

$\{i \geq 0 \ \wedge \ n = dim(a) \ \wedge \ n \geq 1\}$ φ_{init}

SeqInit

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

$\{\exists j \ (0 \leq j \ \wedge \ j+1 < n \ \wedge \ a[j] \geq a[j+1])\}$ $\varphi_{error} \equiv \neg\psi$

CLP encoding of incorrectness

```
incorrect :- initConf(X), reach(X).
reach(Y) :- tr(X,Y), reach(X).
reach(Y) :- errorConf(Y).
initConf(cf(firstCmd, [[int(i), I], [int(n), N], [array(a), A]])
        :- I ≥ 0, dim(A, N), N ≥ 1.                    | φinit
errorConf(cf(haltCmd, [[int(i), I], [int(n), N], [array(a), A]])
        :- 0 ≤ J, J+1 < N, J1 = J+1, AJ ≥ AJ1,         | φerror
           read(A, J, AJ), read(A, J1, AJ1).           |
```

CLP with array constraints

Array constraints

- $\mathtt{read}(\mathtt{a}, \mathtt{i}, \mathtt{v})$ (the $\mathtt{i}$-th element of array $\mathtt{a}$ is $\mathtt{v}$)
- $\mathtt{write}(\mathtt{a}, \mathtt{i}, \mathtt{v}, \mathtt{b})$
  (array $\mathtt{b}$ is equal to array $\mathtt{a}$ except that its $\mathtt{i}$-th element is $\mathtt{v}$)
- $\mathtt{dim}(\mathtt{a}, \mathtt{n})$ (the dimension of $\mathtt{a}$ is $\mathtt{n}$)

Theory of Arrays $\mathcal{A}$

Array congruence

(AC) $\mathtt{I} = \mathtt{J},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}) \ \rightarrow\ \mathtt{U} = \mathtt{V}$

Read-over-Write

(RoW1) $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}) \ \rightarrow\ \mathtt{U} = \mathtt{V}$

(RoW2) $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A}, \mathtt{I}, \mathtt{U}, \mathtt{B}),\ \mathtt{read}(\mathtt{B}, \mathtt{J}, \mathtt{V}) \ \rightarrow\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$

The Constraint Replacement Rule

R5. Constraint Replacement :

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

Then replace `H :- c0, d, G`

by `H :- c1, d, G, ..., H :- cn, d, G`

Constraint Replacements using the Theory of Arrays (1)

Array congruence

(AC) $\mathtt{I} = \mathtt{J},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}) \ \rightarrow\ \mathtt{U} = \mathtt{V}$

[AC1] replace: $\mathtt{I} = \mathtt{J},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$
by: $\mathtt{I} = \mathtt{J},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{U} = \mathtt{V}$

[AC2] replace: $\mathtt{U} \neq \mathtt{V},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V})$
by: $\mathtt{U} \neq \mathtt{V},\ \mathtt{read}(\mathtt{A}, \mathtt{I}, \mathtt{U}),\ \mathtt{read}(\mathtt{A}, \mathtt{J}, \mathtt{V}),\ \mathtt{I} \neq \mathtt{J}$

Read-over-Write

(RoW1) $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}) \ \rightarrow\ \mathtt{U} = \mathtt{V}$

(RoW2) $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V}) \ \rightarrow\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$

[RoW1] replace: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$
by: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{U} = \mathtt{V}$

[RoW2] replace: $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$
by: $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$

[RoW12] replace: $\mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{B},\mathtt{J},\mathtt{V})$
by: $\mathtt{I} = \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{U} = \mathtt{V}$
and $\mathtt{I} \neq \mathtt{J},\ \mathtt{write}(\mathtt{A},\mathtt{I},\mathtt{U},\mathtt{B}),\ \mathtt{read}(\mathtt{A},\mathtt{J},\mathtt{V})$

The Transformation Strategy with Constraint Replacement

Transform(P)

TransfP $= \emptyset$;
Defs $= \{$`incorrect :- initConf(X), reach(X)`$\}$;
while $\exists cl \in$ Defs **do**
  Cls $=$ Unfold(cl);
  Cls $=$ ConstraintReplacement(Cls);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $-$ $\{cl\}$) $\cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

The Transformation Strategy with Constraint Replacement

Transform(P)

TransfP $=\emptyset$;
Defs $=$ {`incorrect :- initConf(X), reach(X)`};
while $\exists cl \in$ Defs **do**
  Cls $=$ Unfold(cl);
  Cls $=$ ConstraintReplacement(Cls);
  Cls $=$ ClauseRemoval(Cls);
  Defs $=$ (Defs $-$ {cl}) $\cup$ Define(Cls);
  TransfP $=$ TransfP $\cup$ Fold(Cls, Defs);
od

Theorem (Termination and Correctness of the Transformation Strategy)

- *Transform*(P) *terminates for all* P;
- `incorrect` $\in M$(P) *iff* `incorrect` $\in M$(TransfP)

Applying the Transformation Strategy

Generation of Verification Conditions;
Reversal;
Propagation of the Error Property.

Transformed VCs for *SeqInit*

```
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), new2(I,N,A).
new2(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), new2(I,N,A).
```

No constrained facts: the program *SeqInit* is correct.

Experimental Evaluation: Array Programs

Program	Gen_W	Gen_{WD}	Gen_S	Gen_{SD}
init	*unknown*	0.06	0.10	0.08
init-partial	*unknown*	0.06	0.07	0.08
init-non-constant	*unknown*	0.06	0.22	0.22
init-sequence	*unknown*	0.80	*unknown*	1.20
copy	*unknown*	0.27	0.33	0.29
copy-partial	*unknown*	0.29	0.34	0.34
copy-reverse	*unknown*	0.27	0.46	0.45
max	*unknown*	0.31	0.24	0.33
sum	*unknown*	0.68	1.14	1.12
difference	*unknown*	0.66	1.15	1.11
find	0.25	0.43	0.46	0.45
first-not-null	0.38	0.41	0.42	0.42
find-first-non-null	1.24	1.87	1.94	1.93
partition	0.06	0.11	0.14	0.12
insertionsort-inner	0.21	0.26	0.45	0.43
bubblesort-inner	2.46	2.71	2.45	2.75
selectionsort-inner	7.20	6.40	7.23	7.16
precision	7	17	16	17
total time	11.80	15.65	17.14	18.48
average time	1.69	0.92	1.07	1.09

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 is the GCD of m and n
```

Initial and error properties

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

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

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

```
incorrect :- initConf(X), reach(X).
reach(Y) :- tr(X,Y), reach(X).
reach(Y) :- errorConf(Y).
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 VCs; Reversal; Propagation of $\varphi_{error}(m,n,z)$

Transformed GCD

```
incorrect :- M>=1,N>=1,M>N,X1=M-N,Z≠D,  new2(M,N,X1,N,Z,D).
incorrect :- M>=1,N>=1,M<N,Y1=N-M,Z≠D,new2(M,N,M,Y1,Z,D).
new2(M,N,X,Y,Z,D) :- M>=1,N>=1,X>Y,X1=X-Y,Z≠D,new2(M,N,X1,Y,Z).
new2(M,N,X,Y,Z,D) :- M>=1,N>=1,X<Y,Y1=Y-X,Z≠D,new2(M,N,X,Y1,Z).
```

No constrained fact: The *gcd* program is correct.

Why Use CLP Transformation for Verification?

- CLP transformation can be used both for *generating* VCs and for *proving* their satisfiability

Why Use CLP Transformation for Verification?

- CLP transformation can be used both for *generating* VCs and for *proving* their satisfiability
- CLP transformation is *parametric* with respect to:
  - programming language and its operational semantics
  - properties and proof rules
  - theory of data structures

Why Use CLP Transformation for Verification?

- CLP transformation can be used both for *generating* VCs and for *proving* their satisfiability

- CLP transformation is *parametric* with respect to:
  - programming language and its operational semantics
  - properties and proof rules
  - theory of data structures

- The input and the output of transformation are semantically equivalent CLP programs.
  - *incremental verification*
  - *composition* of transformations for refining verification

Future Work

- Recursive functions
- More data structure theories (lists, heaps, etc.)
- Other programming languages, properties, proof rules

Thanks for your attention!

Try the VeriMAP tool `http://map.uniroma2.it/VeriMAP`