Verification of Imperative Programs through Transformation of Constraint Logic Programs

Emanuele De Angelis\textsuperscript{1}, 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

LSV-ENS Cachan, Feb 12, 2014
Given the program \textit{prog}:

\[
\begin{align*}
\texttt{x} &= 0; \quad \texttt{y} = 0; \\
\textbf{while} \ (\texttt{x} < \texttt{n}) \ \{ \texttt{x} = \texttt{x} + 1; \quad \texttt{y} = \texttt{y} + 2 \} \\
\end{align*}
\]

and the \textit{specification}:

\[
\{ \texttt{n} \geq 1 \} \ 	exttt{prog} \ \{ \texttt{y} > \texttt{x} \}
\]
Given the program prog:

\[ x = 0; \quad y = 0; \]

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

and the specification:

\[
\{ n \geq 1 \} \text{ prog } \{ y > x \}
\]

Generate the verification conditions (VCs):

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

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

\[ P(x, y, n) \equiv (x = 0 \land y = 0 \land n \geq 1) \lor y > x \]

makes the VCs true

1'. \( x = 0 \land y = 0 \land n \geq 1 \rightarrow (x = 0 \land y = 0 \land n \geq 1) \lor y > x \)

2'. \( ((x = 0 \land y = 0 \land n \geq 1) \lor y > x) \land x < n \)

\rightarrow \( (x + 1 = 0 \land y + 2 = 0 \land n \geq 1) \lor y + 2 > x + 1 \)

3'. \( ((x = 0 \land y = 0 \land n \geq 1) \lor y > x) \land x \geq n \rightarrow y > x \)

and hence the specification \( \{ n \geq 1 \} \text{ prog } \{ y > x \} \) is valid.
The interpretation

\[ P(x, y, n) \equiv (x=0 \land y=0 \land n \geq 1) \lor y > x \]

makes the VCs true

1'. \( x=0 \land y=0 \land n \geq 1 \rightarrow (x=0 \land y=0 \land n \geq 1) \lor y > x \)

2'. \( ((x=0 \land y=0 \land n \geq 1) \lor y > x) \land x < n \rightarrow (x+1=0 \land y+2=0 \land n \geq 1) \lor y + 2 > x + 1 \)

3'. \( ((x=0 \land y=0 \land n \geq 1) \lor y > x) \land x \geq n \rightarrow y > x \)

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

Problem: How to find the interpretation for P automatically?
The VCs are a set of *Horn clauses with constraints*
The VCs are a set of *Horn clauses with constraints*

or, equivalently, a *constraint logic program*:

1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$
2. $P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n)$
3. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow false$

VCs *satisfiable* iff *false* *not* in the *least model* of $V$. 
The VCs are a set of *Horn clauses with constraints*

or, equivalently, a *constraint logic program*:

1. \( x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n) \)
2. \( P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n) \)
4. \( P(x, y, n) \land x \geq n \land y \leq x \rightarrow false \)

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. \( x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n) \)  \textit{Constrained fact}
2. \( P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n) \)
3. \( P(x, y, n) \land x \geq n \land y \leq x \rightarrow \text{false} \)

and derive the \textit{equisatisfiable} V':

4. \( Q(x, y, n) \land x < n \land x > y \land y \geq 0 \rightarrow Q(x + 1, y + 2, n) \)
5. \( Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \text{false} \)
A Transformation-based Method

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

1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$ \textit{Constrained fact}

2. $P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n)$

4. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow \text{false}$

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. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \text{false}$

*No constrained facts*: $V'$ satisfiable with $Q(x, y, n) \equiv \text{false}$. 
A Transformation-based Method

- Apply to V transformations that preserve the least model:
  1. \( x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n) \)  
     \textit{Constrained fact}
  2. \( P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n) \)
  4. \( P(x, y, n) \land x \geq n \land y \leq x \rightarrow \text{false} \)

and derive the \textit{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. \( Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \text{false} \)

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

- Problem: How to transform \( V \) into \( V' \) automatically?
A Transformation-based Method

- Apply to $V$ transformations that *preserve the least model*:
  1. $x = 0 \land y = 0 \land n \geq 1 \rightarrow P(x, y, n)$  
     \textit{Constrained fact}
  2. $P(x, y, n) \land x < n \rightarrow P(x + 1, y + 2, n)$
  4. $P(x, y, n) \land x \geq n \land y \leq x \rightarrow \text{false}$

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. $Q(x, y, n) \land x \geq n \land x \geq y \land y \geq 0 \land n \geq 1 \rightarrow \text{false}$

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

- 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

Experimental evaluation: The VeriMAP system
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
A CLP clause is an implication \( c \land G \rightarrow H \), written as:

\[
H :- c, G.
\]

where \( H \) is an atom, \( c \) is a constraint, and \( G \) is a conjunction of atoms.
A CLP clause is an implication $c \land G \rightarrow H$, written as:

$$H :\leftarrow c, G.$$ 

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 clause is an implication $c \land G \rightarrow H$, written as:

$$H :\leftarrow c, G.$$  

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
A CLP clause is an implication $c \land G \rightarrow H$, 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.
We consider an imperative language with integer variables, assignment, if-else, while-loop, and goto.

Program *increase*:

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

Partial Correctness Specification

\[ \{ x = 0 \land y = 0 \} \text{ increase } \{ x \leq y \} \]
A program is represented as a set of atoms \texttt{at(label, command)}. 

Program \textit{increase}: 

\begin{align*}
\ell_0 &: \quad \text{while}(x < n) \\ \ell_1 &: \quad x = x + 1 \\ \ell_2 &: \quad y = x + y \\ \ell_3 &: \quad \}
\end{align*}

CLP encoding of \textit{increase}:

\begin{align*}
\texttt{at(}\ell_0, \texttt{ite(less(int(x), int(n)), }\ell_1, \ell_h)) &. \\
\texttt{at(}\ell_1, \texttt{asgn(int(x), plus(int(x), int(1))))} &. \\
\texttt{at(}\ell_2, \texttt{asgn(int(y), plus(int(x), int(y))))} &. \\
\texttt{at(}\ell_3, \texttt{goto(}\ell_0)) &. \\
\texttt{at(}\ell_h, \texttt{halt}) &. 
\end{align*}
A transition semantics is defined by:

- a set of configurations, i.e., a CLP term: \( cf(C, S) \)
  where:
  - \( C \) is a labeled command
  - \( S \) is a store, i.e., a list of [variable identifier, value] pairs:
    \[
    \left[ \left[ \text{int}(x), 2 \right], \left[ \text{int}(y), 3 \right] \right]
    \]

- a transition relation: \( tr(cf(C, S), cf(C_1, S_1)) \)
CLP encoding of the operational semantics (2)

<table>
<thead>
<tr>
<th>L: ( \text{Id} = \text{Expr} )</th>
<th>( \text{tr}( \text{cf(cmd(L,asgn(Id,Expr)),S)}, \text{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) { goto L1:</td>
<td>( \text{tr}( \text{cf(cmd(L,ite(Expr,L1,L2)),S)}, \text{cf(C,S))} ) :- beval(Expr,S), at(L1,C).</td>
</tr>
<tr>
<td>} else goto L2</td>
<td>( \text{tr}( \text{cf(cmd(L,ite(Expr,L1,L2)),S)}, \text{cf(C,S))} ) :- beval(not(Expr),S), at(L2,C).</td>
</tr>
<tr>
<td>L: goto L1</td>
<td>( \text{tr}( \text{cf(cmd(L,goto(L1)),S)}, \text{cf(C,S))} ) :- at(L1,C).</td>
</tr>
</tbody>
</table>
Given the specification $\{\varphi_{init}\} \; prog \; \{\psi\}$ define $\varphi_{error} \equiv \neg\psi$

**Definition (Program Incorrectness)**

A program $prog$ is **incorrect** w.r.t. $\varphi_{init}$ and $\varphi_{error}$ if from an initial configuration satisfying $\varphi_{init}$ it is possible to reach a final configuration satisfying $\varphi_{error}$. Otherwise, program $prog$ is **correct**.
Given the specification \( \{ \varphi_{init} \} \ prog \ \{ \psi \} \) define \( \varphi_{error} \equiv \neg \psi \)

**Definition (Program Incorrectness)**

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

Otherwise, program \( prog \) is *correct*.

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

\[
\text{incorrect} :- \text{initConf}(X), \text{reach}(X).
\text{reach}(X) :- \text{tr}(X,Y), \text{reach}(Y).
\text{reach}(X) :- \text{errorConf}(X).
\text{initConf}(X) \equiv X \text{ is a configuration satisfying } \varphi_{init}
\text{errorConf}(X) \equiv X \text{ is a configuration satisfying } \varphi_{error}
\]
CLP encoding of (in)correctness

Given the specification \{ϕ_{init}\} prog \{ψ\} define ϕ_{error} \equiv \negψ

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). \quad \text{reachability}
reach(X) :- errorConf(X).
initConf(X) \equiv X \text{ is a configuration satisfying } ϕ_{init}
errorConf(X) \equiv X \text{ is a configuration satisfying } ϕ_{error}

Theorem (Correctness of Encoding)
prog is correct iff incorrect \notin M(\text{Int}) (the least model of Int)
Partial Correctness Specification

\[
\begin{align*}
\{ x = 0 \land y = 0 \} & \quad \varphi_{\text{init}} \\
\text{increase} & \\
\{ x \leq y \} & \quad \psi \\
\{ x > y \} & \quad \varphi_{\text{error}} \equiv \neg \psi
\end{align*}
\]
Running Example: *increase* (Cont’d)

### Partial Correctness Specification

- \( \{x = 0 \land y = 0\} \quad \phi_{\text{init}} \)
- \( \{x \leq y\} \quad \psi \)
- \( \{x > y\} \quad \phi_{\text{error}} \equiv \neg \psi \)

### Initial and Error Configurations

- \( \text{initConf(cf(cmd(0,ite(...)), [[int(x),X],[int(y),Y],[int(n),N]])) :- X=0, Y=0. \quad \phi_{\text{init}} \)
- \( \text{errorConf(cf(cmd(h,halt), [[int(x),X],[int(y),Y],[int(n),N]])) :- X>Y. \quad \phi_{\text{error}} \)
The Transformation-based Verification Method

Interpreter: \textit{Int}

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

Verification Conditions: \textit{VCs}

Propagate $\varphi_{init}$ or $\varphi_{error}$

\begin{itemize}
  \item \textit{prog correct} if no constrained facts appear in the \textit{VCs}.
  \item \textit{prog incorrect} if the fact \textit{incorrect} appears in the \textit{VCs}.
\end{itemize}
Unfold/Fold Program Transformation

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

- transformation rules:
  \[ R \in \{ \text{Definition, Unfolding, Folding, Clause Removal} \} \]

The transformation rules preserve the least model:
\[ \text{incorrect} \in M(P) \text{ iff } \text{incorrect} \in M(\text{TransfP}) \]

The rules must be guided by a strategy.
Unfold/Fold Program Transformation

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

- transformation rules:
  \[ R \in \{ \text{Definition, Unfolding, Folding, Clause Removal} \} \]

- the transformation rules preserve the least model:
  \[ \text{incorrect} \in M(P) \iff \text{incorrect} \in M(\text{TransfP}) \]
Unfold/Fold Program Transformation

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

- transformation rules:
  \[ R \in \{ \text{Definition, Unfolding, Folding, Clause Removal} \} \]

- the transformation rules preserve the least model:
  \[
  \text{incorrect} \in M(P) \iff \text{incorrect} \in M(\text{TransfP})
  \]

- the rules must be guided by a strategy.
Rules for Transforming CLP Programs

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

\[
\text{newp}(X) :\ - c, A
\]
R1. **Definition.** Introducing a new predicate (e.g., a loop invariant)

\[
\text{newp}(X) : \leftarrow c, A
\]

R2. **Unfolding.** A symbolic evaluation step (resolution)

**given**

\[
H : \leftarrow c, A, G
A : \leftarrow d_1, G_1, \ldots, A : \leftarrow d_m, G_m
\]

**derive**

\[
H : \leftarrow c, d_1, G_1, G, \ldots, H : \leftarrow c, d_m, G_m, G
\]
Rules for Transforming CLP Programs

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

\[ \text{newp}(X) :- c, A \]

R2. Unfolding. A symbolic evaluation step (resolution)

\[
\begin{align*}
\text{given} & \quad H :- c, A, G \\
& \quad A :- d_1, G_1, \ldots, A :- d_m, G_m \\
\text{derive} & \quad H :- c, d_1, G_1, G, \ldots, H :- c, d_m, G_m, G
\end{align*}
\]

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

\[
\begin{align*}
\text{given} & \quad H :- d, A, G \\
& \quad \text{newp}(X) :- c, A \quad \text{and} \quad d \rightarrow c \\
\text{derive} & \quad H :- d, \text{newp}(X), G
\end{align*}
\]
Rules for Transforming CLP Programs

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

\[ \text{newp}(X) :- c, A \]

R2. Unfolding. A symbolic evaluation step (resolution)

\[
\begin{align*}
\text{given} & \quad H :- c, A, G \\
 & \quad A :- d_1, G_1, \ldots, A :- d_m, G_m \\
\text{derive} & \quad H :- c, d_1, G_1, G, \ldots, H :- c, d_m, G_m, G
\end{align*}
\]

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

\[
\begin{align*}
\text{given} & \quad H :- d, A, G \\
 & \quad \text{newp}(X) :- c, A \quad \text{and} \quad d \rightarrow c \\
\text{derive} & \quad H :- d, \text{newp}(X), G
\end{align*}
\]

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

Transform(P)

TransfP = ∅;
Defs = \{incorrect :- initConf(X), reach(X)\};
while ∃ c/ ∈Defs do
    Cls = Unfold(c/);
    Cls = ClauseRemoval(Cls);
   Defs = (Defs − \{c/\}) ∪ Define(Cls);
    TransfP = TransfP ∪ Fold(Cls,Defs);
od

Theorem (Termination and Correctness of the Transformation Strategy)
Transform(P) terminates for all P;
incorrect ∈ M(P) iff incorrect ∈ M(TransfP)
The Transformation Strategy

Transform($P$)

TransfP = ∅;
Defs = {incorrect :- initConf(X), reach(X)};
while ∃cl ∈Defs do
    Cls = Unfold(cl);
    Cls = ClauseRemoval(Cls);
   Defs = (Defs − {cl}) ∪ Define(Cls);
    TransfP = TransfP ∪ Fold(Cls,Defs);
  od

Theorem (Termination and Correctness of the Transformation Strategy)

• Transform($P$) terminates for all $P$;
• incorrect ∈ $M(P)$ iff incorrect ∈ $M(TransfP)$
The most critical transformation step during the unfold/fold transformation strategy is the *introduction of new predicate definitions* to be used for folding.

\[
\text{Given: } \text{p}(X) \text{ :- } c(X,Y), \text{q}(Y).
\]

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

where \( c(X,Y) \rightarrow d(Y) \) (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.
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 \( \text{newp}(Y) :- d(Y), q(Y). \)

where \( c(X,Y) \rightarrow d(Y) \) (\( d(Y) \) is a *generalization* of \( c(X,Y) \))

and *fold*:

\[ p(X) :- c(X,Y), \text{newp}(Y). \]
The most critical transformation step during the unfold/fold transformation strategy is the \textit{introduction of new predicate definitions} to be used for folding.

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

Introduce \( \text{newp}(Y) :- d(Y), q(Y). \)

where \( c(X,Y) \rightarrow d(Y) \) (\( d(Y) \) is a \textit{generalization} of \( c(X,Y) \))

and \textit{fold}: \( p(X) :- c(X,Y), \text{newp}(Y). \)

Generalization strategies based on \textit{widening} and \textit{convex-hull} of linear constraints.
The *specialization* of \( Int \) w.r.t. \( prog \) removes all references to:

- \( \tau \) (i.e., the operational semantics of the imperative language)
- \( \text{at} \) (i.e., the encoding of \( prog \))
The specialization of Int w.r.t. prog removes all references to:
- \( \text{tr} \) (i.e., the operational semantics of the imperative language)
- \( \text{at} \) (i.e., the encoding of prog)

The Specialized Interpreter for increase (Verification Conditions)

- \( \text{incorrect} \) :- \( X = 0 \), \( Y = 0 \), \( \text{new1}(X,Y,N) \).
- \( \text{new1}(X,Y,N) \) :- \( X < N \), \( X1 = X + 1 \), \( Y1 = X1 + Y \), \( \text{new1}(X1,Y1,N) \).
- \( \text{new1}(X,Y,N) \) :- \( X \geq N \), \( X > Y \).
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)

\[
\text{incorrect} :- X = 0, Y = 0, \text{new1}(X, Y, N).
\]
\[
\text{new1}(X, Y, N) :- X < N, X1 = X + 1, Y1 = X1 + Y, \text{new1}(X1, Y1, N).
\]
\[
\text{new1}(X, Y, N) :- X \geq N, X > Y.
\]

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

```prolog
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*:
  ```prolog
  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`. 

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

<table>
<thead>
<tr>
<th>Specialized Verification Conditions for <em>increase</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>...propagating the constraint $X=0$, $Y=0$.</td>
</tr>
<tr>
<td>incorrect :- $N &gt; 0$, $X_1 = 1$, $Y_1 = 1$, new2($X_1$, $Y_1$, $N$).</td>
</tr>
<tr>
<td>new2($X$, $Y$, $N$) :- $X = 1$, $Y = 1$, $N &gt; 1$, $X_1 = 2$, $Y_1 = 3$, new3($X_1$, $Y_1$, $N$).</td>
</tr>
<tr>
<td>new3($X$, $Y$, $N$) :- $X_1 \geq 1$, $Y_1 \geq X_1$, $X &lt; N$, $X_1 = X + 1$, $Y_1 = X_1 + Y$, new3($X_1$, $Y_1$, $N$).</td>
</tr>
<tr>
<td>new3($X$, $Y$, $N$) :- $Y \geq 1$, $N &gt; 0$, $X \geq N$, $X &gt; Y$.</td>
</tr>
</tbody>
</table>

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 :- \( X=0, Y=0, \text{new1}(X,Y,N) \).
2. new2\((X,Y,N) :- X=1, Y=1, N>0, \text{new1}(X,Y,N) \).

Candidate new definition:

\( \text{new3}(X_r,Y_r,N_r) :- X_r=1, Y_r=1, X=2, Y=3, N>1, \text{new1}(X,Y,N) \).

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

Generalization (based on widening):

3. new3\((X,Y,N) :- X \geq 1, Y \geq 1, N>0, \text{new1}(X,Y,N) \).
Program Reversal

P:
\[
\begin{align*}
\text{incorrect} & :\neg a(X), p(X) . \\
p(X) & :- c(X,Y), p(Y) . \\
p(X) & :- b(X) . 
\end{align*}
\]

RevP:
\[
\begin{align*}
\text{incorrect} & :\neg b(X), p(X) . \\
p(Y) & :- c(X,Y), p(X) . \\
p(X) & :- a(X) . 
\end{align*}
\]

\[
\text{incorrect} \in M(P) \text{ iff } \text{incorrect} \in M(\text{RevP})
\]
Propagation of $\varphi_{error}$

### Specialized Verification Conditions for $increase$

**incorrect** :- $N > 0$, $X1 = 1$, $Y1 = 1$, new2($X1$, $Y1$, $N$).

new2($X$, $Y$, $N$) :- $X = 1$, $Y = 1$, $N > 1$, $X1 = 2$, $Y1 = 3$, new3($X1$, $Y1$, $N$).

new3($X$, $Y$, $N$) :- $X1 \geq 1$, $Y1 \geq X1$, $X < N$, $X1 = X + 1$, $Y1 = X1 + Y$, new3($X1$, $Y1$, $N$).

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

### Reversed VCs

**incorrect** :- $Y \geq 1$, $N > 0$, $X \geq N$, $X > Y$, new3($X$, $Y$, $N$).

new3($X1$, $Y1$, $N$) :- $X1 \geq 1$, $Y1 \geq X1$, $X < N$, $X1 = X + 1$, $Y1 = X1 + Y$, new3($X$, $Y$, $N$).

new3($X1$, $Y1$, $N$) :- $X = 1$, $Y = 1$, $N > 1$, $X1 = 2$, $Y1 = 3$, new2($X$, $Y$, $N$).

new2($X1$, $Y1$, $N$) :- $N > 0$, $X1 = 1$, $Y1 = 1$.

### Specialized VCs

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

**incorrect** :- $Y \geq 1$, $N > 0$, $X \geq N$, $X > Y$, 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.

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

- ARMC [Podelski, Rybalchenko PADL 2007]
- HSF(C) [Grebenshchikov et al. TACAS 2012]
- TRACER [Jaffar, Murali, Navas, Santosa CAV 2012]
Improving Precision by Iteration
An Example: Array Initialization.

Program *SeqInit*

\[
i=1; \\
\text{while}(i < n) \{ \\
    a[i] = a[i-1] + 1; \\
    i = i + 1; \\
\}
\]
An Example: Array Initialization.

**Program SeqInit**

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

**An Execution**

```
[4, _, _,_]  \implies [4, 5, _,_]  \implies [4, 5, 6, _]  \implies [4, 5, 6, 7]
```

**Partial Correctness Specification**

\{ i \geq 0 \land n = \text{dim}(a) \land n \geq 1 \} \implies \text{SeqInit} \implies \forall j (0 \leq j \land j + 1 < n \rightarrow a[j] < a[j+1])
An Example: Array Initialization.

Program SeqInit

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

An Execution

\[
[4, _, _, _] \rightarrow [4, 5, _, _] \rightarrow [4, 5, 6, _] \rightarrow [4, 5, 6, 7]
\]

Partial Correctness Specification

\[
\{i \geq 0 \land n = \text{dim}(a) \land n \geq 1\} \quad \text{SeqInit} \\
\{\forall j \ (0 \leq j \land j + 1 < n \rightarrow a[j] < a[j+1])\}
\]
The transition for *array assignment*

- **command**: \( L : a[ie] = e \)
- **store**: \( S \)
- **transition**:

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

Source configuration

Target configuration

Evaluate index expr

Evaluate expression

Get array from store

Update array

Update store

Next label

Next command
Running Example: Array Initialization (Cont’d)

Partial Correctness Specification

\[
\{ i \geq 0 \land n = \text{dim}(a) \land n \geq 1 \} \quad \phi_{\text{init}}
\]

SeqInit

\[
\{ \forall j \ (0 \leq j \land j + 1 < n \rightarrow a[j] < a[j + 1]) \} \quad \psi
\]

\[
\{ \exists j \ (0 \leq j \land j + 1 < n \land a[j] \geq a[j + 1]) \} \quad \phi_{\text{error}} \equiv \neg \psi
\]

CLP encoding of incorrectness

\[
\text{incorrect} :- \text{initConf}(X), \text{reach}(X).
\]

\[
\text{reach}(Y) :- \text{tr}(X, Y), \text{reach}(X).
\]

\[
\text{reach}(Y) :- \text{errorConf}(Y).
\]

\[
\text{initConf}(\text{cf}(\text{firstCmd}, [[\text{int}(i), I], [\text{int}(n), N], [\text{array}(a), A]]))
\]

\[
:\text{:- I} \geq 0, \text{dim}(A, N), N \geq 1. \quad | \quad \phi_{\text{init}}
\]

\[
\text{errorConf}(\text{cf}(\text{haltCmd}, [[\text{int}(i), I], [\text{int}(n), N], [\text{array}(a), A]]))
\]

\[
:\text{:- 0} \leq J, J + 1 < N, J1 = J + 1, AJ \geq AJ1, \quad | \quad \phi_{\text{error}}
\]

\[
\text{read}(A, J, AJ), \text{read}(A, J1, AJ1).
\]
Array constraints

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

Theory of Arrays $\mathcal{A}$

Array congruence

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

Read-over-Write

(RoW1)  $I = J$, write(A, I, U, B), read(B, J, V) $\rightarrow$ U = V
(RoW2)  $I \neq J$, write(A, I, U, B), read(B, J, V) $\rightarrow$ read(A, J, V)
R5. Constraint Replacement:

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 $H :\neg c_0$, d, G

by $H :\neg c_1$, d, G, ..., $H :\neg c_n$, d, G
Array congruence

\[(AC) \quad I = J, \, \text{read}(A, I, U), \, \text{read}(A, J, V) \rightarrow U = V\]

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

[AC2] replace: \(U \neq V, \, \text{read}(A, I, U), \, \text{read}(A, J, V)\)
by: \(U \neq V, \, \text{read}(A, I, U), \, \text{read}(A, J, V), \, I \neq J\)
### Constraint Replacements using the Theory of Arrays (2)

#### Read-over-Write

<table>
<thead>
<tr>
<th>(RoW1)</th>
<th>I = J, write(A, I, U, B), read(B, J, V) → U = V</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RoW2)</td>
<td>I ≠ J, write(A, I, U, B), read(B, J, V) → read(A, J, V)</td>
</tr>
</tbody>
</table>

**[RoW1]**
- replace: I = J, write(A, I, U, B), read(B, J, V)
- by: I = J, write(A, I, U, B), U = V

**[RoW2]**
- replace: I ≠ J, write(A, I, U, B), read(B, J, V)
- by: I ≠ J, write(A, I, U, B), read(A, J, V)

**[RoW12]**
- replace: write(A, I, U, B), read(B, J, V)
- by: I = J, write(A, I, U, B), U = V
- and I ≠ J, write(A, I, U, B), read(A, J, V)
### Transform($P$)

<table>
<thead>
<tr>
<th>Transform Strategy with Constraint Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransfP = $\emptyset$;</td>
</tr>
<tr>
<td>Defs = { <strong>incorrect</strong> :- initConf(X), reach(X) };</td>
</tr>
<tr>
<td><strong>while</strong> ( \exists cl \in \text{Defs} ) <strong>do</strong></td>
</tr>
<tr>
<td>\hspace{1em} Cls = Unfold(cl);</td>
</tr>
<tr>
<td>\hspace{1em} Cls = ConstraintReplacement(Cls);</td>
</tr>
<tr>
<td>\hspace{1em} Cls = ClauseRemoval(Cls);</td>
</tr>
<tr>
<td>\hspace{1em} Defs = (Defs \setminus {cl}) \cup Define(Cls);</td>
</tr>
<tr>
<td>\hspace{1em} TransfP = TransfP \cup Fold(Cls, Defs);</td>
</tr>
<tr>
<td><strong>od</strong></td>
</tr>
</tbody>
</table>

**Theorem (Termination and Correctness of the Transformation Strategy)**

Transform($P$) terminates for all $P$;
incorrect \( \in \mathcal{M}(P) \) iff incorrect \( \in \mathcal{M}(\text{TransfP}) \)
The Transformation Strategy with Constraint Replacement

**Transform**(*P*)

\[
\text{TransfP} = \emptyset; \\
\text{Defs} = \{ \text{incorrect} : - \text{initConf}(X), \text{reach}(X) \}; \\
\textbf{while} \ \exists cI \in \text{Defs} \ \textbf{do} \\
\hspace{1em} \text{Cls} = \text{Unfold}(cI); \\
\hspace{1em} \text{Cls} = \text{ConstraintReplacement}(\text{Cls}); \\
\hspace{1em} \text{Cls} = \text{ClauseRemoval}(\text{Cls}); \\
\hspace{1em} \text{Defs} = (\text{Defs} - \{ cI \}) \cup \text{Define}(\text{Cls}); \\
\hspace{1em} \text{TransfP} = \text{TransfP} \cup \text{Fold}(\text{Cls}, \text{Defs}); \\
\textbf{od}
\]

**Theorem (Termination and Correctness of the Transformation Strategy)**

- **Transform**(*P*) terminates for all *P*;

- incorrect ∈ \( M(P) \) iff incorrect ∈ \( M(\text{TransfP}) \)
Applying the Transformation Strategy

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

Transformed VCs for \textit{SeqInit}

\textbf{incorrect} :- \( J1 = J + 1, J \geq 0, J1 < I, AJ \geq AJ1, D = I - 1, N = I + 1, Y = X + 1, \)
\hspace{1em} read\((A, J, AJ)\), read\((A, J1, AJ1)\), read\((A, D, X)\), write\((A, I, Y, B)\),
\hspace{1em} new1\((I, N, A)\).
\textbf{new1}\((I1, N, B)\) :- \( I1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, N \leq I + 2, \)
\hspace{1em} \quad \quad \quad \quad I \geq 1, Z < I, Z \geq 1, N > I, U \geq V, \quad \quad \text{read}(A, W, U), \quad \text{read}(A, Z, V), \)
\hspace{1em} \quad \quad \quad \quad \text{read}(A, D, X), \quad \text{write}(A, I, Y, B), \quad \text{new2}(I, N, A).
\textbf{new2}\((I1, N, B)\) :- \( I1 = I + 1, Z = W + 1, Y = X + 1, D = I - 1, I \geq 1, \)
\hspace{1em} \quad \quad \quad \quad Z < I, Z \geq 1, N > I, U \geq V, \quad \quad \text{read}(A, W, U), \quad \text{read}(A, Z, V), \)
\hspace{1em} \quad \quad \quad \quad \text{read}(A, D, X), \quad \text{write}(A, I, Y, B), \quad \text{new2}(I, N, A).

No constrained facts: the program \textit{SeqInit} is \textit{correct}. 
# Experimental Evaluation: Array Programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Gen&lt;sub&gt;W&lt;/sub&gt;</th>
<th>Gen&lt;sub&gt;WD&lt;/sub&gt;</th>
<th>Gen&lt;sub&gt;S&lt;/sub&gt;</th>
<th>Gen&lt;sub&gt;SD&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>unknown</td>
<td>0.06</td>
<td>0.10</td>
<td>0.08</td>
</tr>
<tr>
<td>init-partial</td>
<td>unknown</td>
<td>0.06</td>
<td>0.07</td>
<td>0.08</td>
</tr>
<tr>
<td>init-non-constant</td>
<td>unknown</td>
<td>0.06</td>
<td>0.22</td>
<td>0.22</td>
</tr>
<tr>
<td>init-sequence</td>
<td>unknown</td>
<td>0.80</td>
<td>unknown</td>
<td>1.20</td>
</tr>
<tr>
<td>copy</td>
<td>unknown</td>
<td>0.27</td>
<td>0.33</td>
<td>0.29</td>
</tr>
<tr>
<td>copy-partial</td>
<td>unknown</td>
<td>0.29</td>
<td>0.34</td>
<td>0.34</td>
</tr>
<tr>
<td>copy-reverse</td>
<td>unknown</td>
<td>0.27</td>
<td>0.46</td>
<td>0.45</td>
</tr>
<tr>
<td>max</td>
<td>unknown</td>
<td>0.31</td>
<td>0.24</td>
<td>0.33</td>
</tr>
<tr>
<td>sum</td>
<td>unknown</td>
<td>0.68</td>
<td>1.14</td>
<td>1.12</td>
</tr>
<tr>
<td>difference</td>
<td>unknown</td>
<td>0.66</td>
<td>1.15</td>
<td>1.11</td>
</tr>
<tr>
<td>find</td>
<td>0.25</td>
<td>0.43</td>
<td>0.46</td>
<td>0.45</td>
</tr>
<tr>
<td>first-not-null</td>
<td>0.38</td>
<td>0.41</td>
<td>0.42</td>
<td>0.42</td>
</tr>
<tr>
<td>find-first-non-null</td>
<td>1.24</td>
<td>1.87</td>
<td>1.94</td>
<td>1.93</td>
</tr>
<tr>
<td>partition</td>
<td>0.06</td>
<td>0.11</td>
<td>0.14</td>
<td>0.12</td>
</tr>
<tr>
<td>insertionsort-inner</td>
<td>0.21</td>
<td>0.26</td>
<td>0.45</td>
<td>0.43</td>
</tr>
<tr>
<td>bubblesort-inner</td>
<td>2.46</td>
<td>2.71</td>
<td>2.45</td>
<td>2.75</td>
</tr>
<tr>
<td>selectionsort-inner</td>
<td>7.20</td>
<td>6.40</td>
<td>7.23</td>
<td>7.16</td>
</tr>
<tr>
<td>precision</td>
<td>7</td>
<td>17</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>total time</td>
<td>11.80</td>
<td>15.65</td>
<td>17.14</td>
<td>18.48</td>
</tr>
<tr>
<td>average time</td>
<td>1.69</td>
<td>0.92</td>
<td>1.07</td>
<td>1.09</td>
</tr>
</tbody>
</table>
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

ϕ_{init}(m,n) ≡ m ≥ 1 ∧ n ≥ 1

ϕ_{error}(m,n,z) ≡ ∃d (gcd(m, n, d) ∧ d ≠ 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.
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 \( \text{gcd} \) 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