The VeriMAP system for program transformation and verification

Fabio Fioravanti
University of Chieti-Pescara, Italy

joint work with Emanuele De Angelis, Maria Chiara Meo, Alberto Pettorossi and Maurizio Proietti
Outline

- Constrained Horn Clauses (CHC) for verification
- CHC transformation rules and strategies
- Semantics-based translation to CHC
- CHC specialization as CHC solving
- Verification of relational properties (e.g. equivalence, functionality, non-interference)
- Verification of programs with inductively-defined data structures (e.g., lists and trees)
- Verification of time-aware business processes
- VeriMAP demo
Constrained Horn Clauses (CHC)

- **Constrained Horn Clauses** (aka Constraint Logic Programs):
  \[ A_0 \leftarrow c, A_1, \ldots, A_n \]
  where: (1) \( A_0 \) is false or an atom, (2) \( A_1, \ldots, A_n \), \( n \geq 0 \), are atoms, and (3) \( c \) is a constraint in a first order theory \( Th \).
  All variables are assumed to be universally quantified in front.

Many verification problems can be encoded as CHC satisfiability

- **Satisfiability**: Given a set \( P \) of CHC, has \( P \cup Th \) a model?

- **Solving**: Compute a model of \( P \cup Th \), expressed in \( Th \) (if sat) or return unsat; solvability implies satisfiability, not vice versa

- **CHC solvers**: SMT solvers for the Horn fragment with Linear Integer/Real Arithmetic, Booleans, Arrays, Lists, Bit-vectors (e.g., Z3 (SPACER), Eldarica, HSF, MathSAT, Hoice, RAHFT/PECOS, VeriMAP, ...)

- **CHC tools**: Ciao, SeaHorn, ...
Imperative program verification via CHC solving

• Summing the first $n$ integers

Specification
\{n \geq 0\} \ x=0; \ y=0; \ \text{while} \ (x < n) \ \{ \ x=x+1; \ y=x+y \} \ \{y \geq x\}

Constrained Horn Clauses
\begin{align*}
p(X, Y, N) & \leftarrow N \geq 0, \ X=0, \ Y=0 \quad \text{%Init} \\
p(X1, Y1, N) & \leftarrow X < N, \ X1=X+1, \ Y1=X1+Y, \ p(X, Y, N) \quad \text{%Loop} \\
false & \leftarrow X \geq N, \ Y < X, \ p(X, Y, N) \quad \text{%Exit}
\end{align*}

• Solution (i.e., model) of the CHCs:
$p(X, Y, N) \mapsto X \geq 0, \ Y \geq X$

• CHC are solvable, hence satisfiable, and the specification is valid
CHC transformation for verification

- CHC transformations
  - propagate constraints (backward and forward)
    - Unfolding and constraint solving
    - discover inductive invariants (also using widening & convex-hull)
      - Definition and folding
      - discover relations among predicates
  - CHC transformations
    - preserve satisfiability
    - preserve solvability, and can improve it
    - can improve the effectiveness of state-of-the-art CHC solvers
CHC transformation rules and strategies
Transformations of Functional and Logic Programs

Transformation techniques introduced for improving functional and logic programs [Burstall-Darlington 1977, Tamaki-Sato 1984] can be adapted to ease satisfiability proofs for CHCs.

Initial program $P_0 \rightarrow P_1 \rightarrow \ldots \rightarrow P_n$ Final program

where '$\rightarrow$' is an application of a transformation rule.

• Each rule application preserves the semantics:
  $M(P_0) = M(P_1) = \ldots = M(P_n)$

• The application of the rules is guided by a strategy that guarantees that $P_n$ is more efficient than $P_0$. 
Transformation Rules for CHCs

Initial clauses $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ Final clauses

where '$\rightarrow$' is an application of a transformation rule.
Transformation Rules for CHCs

<table>
<thead>
<tr>
<th>Initial clauses</th>
<th>( S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n )</th>
<th>Final clauses</th>
</tr>
</thead>
</table>

where '→' is an application of a transformation rule.

R1. **Definition.** Introduce a new predicate definition

introduce \( C : \text{newp}(X) :- c, G \)

\[ S_{i+1} = S_i \cup \{C\} \quad \text{Defs} := \text{Defs} \cup \{C\} \]
Transformation Rules for CHCs

Initial clauses $S_0 \to S_1 \to \ldots \to S_n$ Final clauses

where $\to$ is an application of a transformation rule.

R1. Definition. Introduce a new predicate definition

introduce $C$: newp(X) :- c, G

$S_{i+1} = S_i \cup \{C\}$ Defs := Defs $\cup \{C\}$

R2. Unfolding. Apply a Resolution step

given $C$: H :- c,A,G A :- d_1,G_1 $\ldots$ A :- d_m,G_m in $S_i$
derive $S = \{ H :- c,d_1,G_1,G \ldots H :- c,d_m,G_m,G \}$

$S_{i+1} = (S_i - \{C\}) \cup S$
Transformation Rules for CHCs

R3. **Folding.** Replace a conjunction with a new predicate

given \( C: H : - d, B, G \) in \( S_i \) \( \text{newp}(X) : - c, B. \) with \( d \rightarrow c \) in Defs

derive \( D: H : - d, \text{newp}(X), G. \)

\[ S_{i+1} = (S_i - \{C\}) \cup \{D\} \]
Transformation Rules for CHCs

R3. **Folding.** Replace a conjunction with a new predicate

given \( C: H :- d, B, G \) in \( S_i \) \( \text{newp}(X) :- c, B \). with \( Th \models d \rightarrow c \) in Defs

derive \( D: H :- d, \text{newp}(X), G \).

\( S_{i+1} = (S_i - \{C\}) \cup \{D\} \)

R4. **Constraint replacement.** Replace a constraint with an equivalent one

given \( C: H :- c, B, G \) in \( S_i \) with \( Th \models c \leftrightarrow d \)

derive \( D: H :- d, B, G \)

\( S_{i+1} = (S_i - \{C\}) \cup \{D\} \)
Transformation Rules for CHCs

R3. Folding. Replace a conjunction with a new predicate

\[
\text{given } C: H :\neg d, B, G \text{ in } S_i \quad \text{newp}(X) :\neg c, B. \quad \text{with } Th \models d \rightarrow c \text{ in Defs}
\]

\[
\text{derive } D: H :\neg d, \text{newp}(X), G.
\]

\[
S_{i+1} = (S_i - \{C\}) \cup \{D\}
\]

R4. Constraint replacement. Replace a constraint with an equivalent one

\[
\text{given } C: H :\neg c, B, G \text{ in } S_i \quad \text{with } Th \models c \leftrightarrow d
\]

\[
\text{derive } D: H :\neg d, B, G
\]

\[
S_{i+1} = (S_i - \{C\}) \cup \{D\}
\]

R5. Clause Removal. Remove a clause C with unsatisfiable constraint or subsumed by another

\[
S_{i+1} = (S_i - \{C\})
\]
Transformation Rules for CHCs

R3. **Folding.** Replace a conjunction with a new predicate

given $C: H :- d, B, G$ in $S_i$  
newp($X$) :- c, B. with $\text{Th} \models d \rightarrow c$ in Defs

derive $D: H :- d, \text{newp}(X), G.$

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R4. **Constraint replacement.** Replace a constraint with an equivalent one

given $C: H :- c, B, G$ in $S_i$ with $\text{Th} \models c \leftrightarrow d$

derive $D: H :- d, B, G$

$S_{i+1} = (S_i - \{C\}) \cup \{D\}$

R5. **Clause Removal.** Remove a clause C with unsatisfiable constraint or subsumed by another

$S_{i+1} = (S_i - \{C\})$

**Theorem** [Tamaki-Sato 84, Etalle-Gabbrielli 96]: If every new definition is unfolded at least once in $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ then

$S_0$ satisfiable  iff  $S_n$ satisfiable
Transformation strategies

• Transformation rules need to be guided by suitable strategies.

• Main idea: exploit some knowledge about the query to produce a customized, easier to verify set of clauses.

• **Specialization** [Gallagher, Leuschel, FPP, ...]: Given a set of clauses $S$ and a query $\text{false} \ :- \ c,A$, where $A$ is atomic, transform $S$ into a set of clauses $S_{SP}$ such that

$$S \cup \{\text{false} \ :- \ c,A\} \text{ satisfiable} \iff S_{SP} \cup \{\text{false} \ :- \ c,A\} \text{ satisfiable}.$$ 

• **Predicate Tupling** (also known as **Conjunctive Partial Deduction**) [PP, Leuschel, ...]: Given a set of clauses $S$ and a query $\text{false} \ :- \ c,G$, where $G$ is a (non-atomic) conjunction, introduce a new predicate $\text{newp}(X) \ :- \ G$ and transform set of clauses $S_{T}$ such that

$$S \cup \{\text{false} \ :- \ c,G\} \text{ satisfiable} \iff S_{T} \cup \{\text{false} \ :- \ c,\text{newp}(X)\} \text{ satisfiable}.$$
Specialization Strategy: An Example

false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X>=0, tm_halts(X).

% ∀X. p(X,b) → X>=0

% the X-th Turing machine halts on X
Specialization Strategy: An Example

false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X>=0, tm_halts(X).

% ∀X. p(X,b) → X>=0
S_0

% the X-th Turing machine halts on X

Define: q(X) :- X<0, p(X,b).
% q(X) is a specialization of p(X,C)
% to a specific constraint on X and value of C
S_1
Specialization Strategy: An Example

\[
\text{false} :- X<0, \ p(X,b). \\
p(X,C) :- X=Y+1, \ p(Y,C). \\
p(X,a). \\
p(X,b) :- X\geq0, \ \text{tm\_halts}(X). \\
\]

% \forall X. \ p(X,b) \rightarrow X\geq0

\% the X-th Turing machine halts on X

Define: \quad q(X) :- X<0, \ p(X,b). \quad \% q(X) is a specialization of p(X,C)

% to a specific constraint on X and value of C

Unfold: \quad q(X) :- X<0, \ X=Y+1, \ p(Y,b). \quad \% clause removal

\quad q(X) :- X<0, \ X\geq0, \ \text{tm\_halts}(X). \quad \% clause removal
Specialization Strategy: An Example

false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X>=0, tm_halts(X).

% ∀X. p(X,b) → X>=0

% the X-th Turing machine halts on X

Define: q(X) :- X<0, p(X,b).
% q(X) is a specialization of p(X,C)
% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b).
q(X) :- X<0, X>=0, tm_halts(X). % clause removal

Fold: false :- X<0, q(X).
q(X) :- X<0, X=Y+1, q(Y).

Satisfiability of S_3 is easy to check: q(X) \equiv false makes all clauses true (no facts for q)
A Generic U/F Transformation Strategy

1. Define
2. Unfold
3. Replace Constraints
4. Remove Clauses

Fold? (no)

S₀

Fold? (yes)

Sₙ
Some Issues About the U/F Strategy

• **Unfolding**: Which atoms should be unfolded? When to stop?

• **Constraint replacement**: A suitable constraint reasoner is needed

• **Definition**: Suitable new predicates need to be introduced to guarantee termination and effectiveness of strategy
  
  - Definitions are arranged in a tree
  
  - New definitions possibly contain a *generalized* constraint
    
    - newp :- d, B  **ancestor definition**
    
    - newp :- c, B  **candidate definition**
    
    - newp :- g, B  **generalized definition**  \( c \rightarrow g = \text{gen}(c,d) \)

  - Generalization operators based on widening and convex-hull  
    [Cousot-Cousot 77, Cousot-Halbwachs 78, Bagnara et al. 08]
Semantics-based translation to CHC
Verification Conditions
**CHC Specialization as a Verification Condition Generator**

The CHC specializer is parametric with respect to the programming language \( L \) and the class of properties.

**Diagram:**
- **Program** \( P \) in \( L \)
- **Property** \( F \)
- \( \text{Interp}_L \)
- **CHC Specializer**
- **VC**: Verification Conditions, i.e., a set of CHCs independent of \( L \)

**Equation:**

\[
F \text{ holds for } P \iff \text{VC is satisfiable}
\]
## Translating Imperative Programs into CHC

- **C-like imperative language** with assignments, conditionals, jumps.
  While-loops translated to conditionals and jumps.

- **Commands encoded as atomic assertions**: \( \text{at}(\text{Label}, \text{Cmd}) \).

| x=0; | 0. x=0; | at(0, asgn(int(x), int(0))). |
| y=0; | 1. y=0; | at(1, asgn(int(y), int(0))). |
| while (x<n) { | 2. if (x<n) 3 else 6; | at(2, ite(less(int(x), int(n)), 3, 6))). |
| x=x+1; | 3. x=x+1; | at(3, asgn(int(x), plus(int(x), int(1))))). |
| y=x+y | 4. y=x+y; | at(4, asgn(int(y), plus(int(x), int(y))))). |
| } | 5. goto 2; | at(5, goto(2)). |
| h. halt | at(h, halt). |
A Small-Step Operational Semantics

- The operational semantics is a **one-step transition relation** between configurations
  \[ <n:cmd, env> \Rightarrow <n’:cmd’, env’> \]
  where:  \( n:cmd \) is a labelled command
  \( env \) is an environment mapping variable identifiers to values

- **Assignment**
  \[ <n: x=e, env> \Rightarrow <\text{next}(n), \text{update}(env, x, [e]env)> \]
  \( \text{next}(n) \) is the next labelled command
  \( \text{update}(env, x, [e]env) \) updates the value of \( x \) to the value of expression \( e \) in \( env \)

- **Conditional**
  \[ <n: \text{if (e) n1 else n2}, env> \Rightarrow <\text{at}(n1), env> \quad \text{if} \ [e]env \neq 0 \]
  \[ <n: \text{if (e) n1 else n2}, env> \Rightarrow <\text{at}(n2), env> \quad \text{if} \ [e]env = 0 \]
  \( \text{at}(n) \) is the labelled command with label \( n \)

- **Jump**
  \[ <n: \text{goto n1}, env> \Rightarrow <\text{at}(n1), env> \]
A CHC Interpreter for the Small-Step Semantics

- **Configurations:** cf(LC, Env)
  where:
  - LC is a labelled command represented as a term of the form cmd(L,C),
    L is a label, C is a command
  - Env is an environment represented as a list of (variable-id,value) pairs:
    [(x,X),(y,Y),(z,Z)]

- **One-step transition relation** between configurations:

  \[ \text{tr}( cf(LC1,Env1), cf(LC2,Env2) ) \]
CHC Interpreter (Asgn)

assignment \ x=e;

source configuration target configuration

\begin{align*}
\text{tr(} & \text{cf(cmd(L, asgn(X,E)), Env1), cf(cmd(L1, C), Env2)} \text{ })::-
\begin{align*}
& \text{nextlab(L,L1),} & \text{next label} \\
& \text{at(L1,C),} & \text{next command} \\
& \text{eval(E,Env1,V),} & \text{evaluate expression} \\
& \text{update(Env1,X,V,Env2).} & \text{update environment}
\end{align*}
\end{align*}

More clauses for predicate \text{tr} to encode the semantics of the other commands.
Encoding Partial Correctness Properties

- **Partial correctness** specification (Hoare triple):
  \[
  \{ \phi \} \text{prog} \{ \psi \}
  \]
  *If* the initial values of the program variables satisfy the precondition \( \phi \) and \text{prog} terminates, *then* the final values of the program variables satisfy the postcondition \( \psi \).

- **CHC encoding** of partial correctness:

  \[
  \begin{align*}
  \text{false} & : \text{initConf}(Cf), \text{errReach}(Cf). \\
  \text{errReach}(Cf) & : \text{errorConf}(Cf). \\
  \text{errReach}(Cf) & : \text{tr}(Cf, Cf2), \text{errReach}(Cf2). \\
  \text{initConf}(cf(C, Env)) & : \text{at}(0, C), \phi(Env). \\
  \text{errorConf}(cf(C, Env)) & : \text{at}(h, C), \neg \psi(Env). \\
  \text{tr}(Cf1, Cf2) & : \ldots
  \end{align*}
  \]

- \( \{ \phi \} \text{prog} \{ \psi \} \) is **valid** iff \( \text{PC-prop} \) is **satisfiable**.
Problems of direct CHC encoding

- **PC-prop** includes a lot of complex structures and predicates:
  - **complex terms** encoding configurations:
    \[
    \text{cf(cmd(L,asgn(X,Expr)),[(x,1),(y,0),(a,[2,3,4])])}
    \]
  - **recursive predicates over lists** encoding functions on the environment:
    \[
    \text{update([(X,N)|Bs],X,V,[(X,V)|Cs]) :- ... update(Bs,X,V,Cs)}
    \]
- State-of-the-art CHC solvers **hardly terminate** when checking the satisfiability of **PC-prop**
VCGen: Generating Verification Conditions

VCGen is a transformation strategy that specializes PC-prop to a given \{ϕ\} prog \{ψ\}, removes explicit reference to the interpreter (function \textit{cf}, predicates \textit{at}, \textit{tr}, etc.).

- All new definitions are of the form newp(X) :- errReach(cf(LC,Env)), corresponding to a program point.
  - Limited reasoning about constraints at specialization time (satisfiability only).

- VCGen is parametric wrt Interp\textsubscript{L} (to a large extent).

- If PC-prop \xrightarrow{\textit{VCGen}} VC then PC-prop is satisfiable iff VC is satisfiable
  - no complex terms or lists occur in VC
Generating Verification Conditions: An Example

PC property:

\{n \geq 1\} \text{SumUpto} \{y > x\}

CHC encoding:

false :- initConf(Cf), errReach(Cf).
errReach(Cf) :- errorConf(Cf).
errReach(Cf1) :- tr(Cf1,Cf2), errReach(Cf2).
initConf(cf(C, [(x,X),(y,Y),(n,N)])) :- at(0,C), N \geq 1.
errorConf(cf(C, [(x,X),(y,Y),(n,N)])) :- at(h,C), Y \leq X.
tr(Cf1,Cf2) :- ...
...
at(0,asgn(int(x), int(0))).
...

VCGen

Verification Conditions:

false :- N \geq 1, X=0, Y=0, p(X, Y, N).
p(X, Y, N) :- X < N, X1=X+1, Y1=Y+2, p(X1, Y1, N).
p(X, Y, N) :- X \geq N, Y \leq X.
Two semantics for function calls

- **Small-Step semantics (SS)**
  - “dives into” the function definition
  - VC are linear clauses (one atom in the body)

- **Multi-Step semantics (MS)**
  - “wraps” the whole function call
  - VC are non-linear
  - `reach(C,C).
    reach(C,C2) :- tr(C,C1), reach(C1,C2).`

  false :- `initConf(C1), reach(C1,C2), errorConf(C2).`

- more variables (use variants of Leuschel’s Redundant Argument Filtering)
Properties of VCGen

• The number of transformation steps is \textit{linear} wrt the size of the imperative program P

• The size of VC (the number of CHC) is \textit{linear} wrt the size of program P
Short demo
Experimental evaluation

- Other semantics: exceptions, etc.
- Checking the satisfiability of the VCs using QARMC, Z3 (PDR), MathSAT (IC3), Eldarica
- VCGen+QARMC compares favorably to HSF+QARMC

<table>
<thead>
<tr>
<th></th>
<th>Small-step ($SS^s$)</th>
<th>Multi-step ($MS$)</th>
<th>HSF(C)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>QARMC</td>
<td>Z3</td>
<td>MSAT</td>
</tr>
<tr>
<td>Correct answers</td>
<td>217</td>
<td>208</td>
<td>205</td>
</tr>
<tr>
<td>safe problems</td>
<td>161</td>
<td>150</td>
<td>158</td>
</tr>
<tr>
<td>unsafe problems</td>
<td>56</td>
<td>58</td>
<td>47</td>
</tr>
<tr>
<td>Incorrect answers</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>false alarms</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>missed bugs</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Timeouts</td>
<td>98</td>
<td>112</td>
<td>112</td>
</tr>
<tr>
<td>Total problems</td>
<td>320</td>
<td>320</td>
<td>320</td>
</tr>
</tbody>
</table>

| VCG time                     | 221.68  | 221.68 | 221.68 | 221.68 | 141.85  | 141.85 | 141.85 | 141.85 | 141.85 |
| Solving time                 | 3656.24 | 4221.39| 2988.86| 8809.58| 2674.00 | 2704.95| 1896.96| 2779.18|        |
| Total time                   | 3877.92 | 4443.07| 3210.54| 9031.26| 2815.85 | 2846.80| 2038.81| 2921.03|        |
| Average Time                 | 17.87   | 21.36  | 15.66  | 41.62  | 13.41   | 14.52  | 11.52  | 16.05  |        |
Comments

- Semantics-based Verification Condition generation is efficient and flexible
- Experiments with C, BPMN (business processes), Erlang (ongoing)
- Future work
  - More language semantics
    - Use formal semantics specifications of the K-Framework [Rosu et al.]
      ANSI C, OCaml, Python, PHP, Java, Javascript, Ethereum Virtual Machine...
  - Make it accessible to third parties
    - improve documentation
- References
  - [DFPP - PPDP 15], [DFPP-ScienceCompProgr 16]
  - http://map.uniroma2.it/VeriMAP
  - http://map.uniroma2.it/vcgen
Short demo
CHC Specialization as CHC Solving
VCTransf: Specializing Verification Conditions

false :- c, p(X)

newp(X) :- c, p(X)

apply theory of constraints

Specializing verification conditions by propagating constraints.

Introduction of new predicates by generalization (e.g., widening and convex hull techniques)

VC is satisfiable iff VC’ is satisfiable

VC

Define

Unfold

Replace Constraints

Remove Clauses

Fold?

no

yes

VC’
VCTransf as CHC Solving

The effect of applying VCTransf can be:

1. A set \( \text{VC}' \) of verification conditions without constrained facts for the predicates on which the queries depend (i.e., no clauses of the form \( p(X) :\neg c \)).
   \( \text{VC}' \) is satisfiable.

2. A set \( \text{VC}' \) of verification conditions including false :\neg true.
   \( \text{VC}' \) is unsatisfiable.

3. Neither 1 nor 2 (constrained facts of the form \( p(X) :\neg c \), but not false :\neg true).
   Satisfiability is unknown.

\[
\begin{align*}
\text{false} & : \text{X<0, p(X,b)}. \quad \text{VC} \\
p(X,C) & : \text{X=Y+1, p(Y,C)}. \\
p(X,a). \\
p(X,b) & : \text{X0, tm_halts(X)}. \\
\end{align*}
\]

\[
\begin{align*}
\text{false} & : \text{X<0, q(X)}. \quad \text{VC'} \\
q(X) & : \text{X<0, X=Y+1, q(Y)}. \\
\end{align*}
\]

No constrained facts: \( \text{VC}' \) satisfiable

propagation of constraint X<0 and constant b
Iterated CHC Specialization

- If the satisfiability of VC’ is unknown, $\text{VCTransf}$ can be iterated.

- Between two applications of $\text{VCTransf}$ we can apply the $\text{Reversal}$ transformation (particular case of the query-answer transformation [KafleGallagher 15] for linear programs) that interchanges premises and conclusions of clauses (backward reasoning from queries simulates forward reasoning from facts).

VC is satisfiable iff VC’ is satisfiable

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

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

$\text{VC}_0 \xrightarrow{\text{VCTransf}} \text{VC}_1 \xrightarrow{\text{Reversal}} \text{VC}_2 \xrightarrow{\text{VCTransf}} \text{VC}_3 \ldots \xrightarrow{\text{VCTransf}} \text{VC}_n$
Iterated CHC Specialization: *SumUpto* Example

\[
\text{false : - } N \geq 1, \ X=0, \ Y=0, \ p(X, Y, N). \\
p(X, Y, N) : - \ X<N, \ X1=X+1, \ Y1=Y+2, \ p(X1, Y1, N). \\
p(X, Y, N) : - \ X \geq N, \ Y<X.
\]
Iterated CHC Specialization: *SumUpto* Example

\[\text{false} :\quad N \geq 1, X=0, Y=0, \ p(X, Y, N).\]

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

\[p(X, Y, N) :\quad X \geq N, Y < X.\]

\[\text{false} :\quad N \geq 1, X1=1, Y1=1, \ \text{new2}(X1, Y1, N).\]

\[\text{new2}(X, Y, N) :\quad X=1, Y=1, N>1, X1=2, Y1=3, \ \text{new3}(X1, Y1, N).\]

\[\text{new3}(X, Y, N) :\quad X1 \geq 1, Y1 \geq X1, X < N, X1=X+1, Y1=X1+Y, \ \text{new3}(X1, Y1, N).\]

\[\text{new3}(X, Y, N) :\quad Y \geq 1, N \geq 1, X \geq N, Y < X.\]
Iterated CHC Specialization: \textit{SumUpto} Example

\begin{itemize}
  \item \textbf{false} :- \texttt{N}\geq 1, X=0, Y=0, \texttt{p(X, Y, N)}.
  \item \texttt{p(X, Y, N)} :- X<N, X1=X+1, Y1=Y+2, \texttt{p(X1, Y1, N)}.
  \item \texttt{p(X, Y, N)} :- X\geq N, Y<X.
  \item \texttt{false} :- \texttt{N}\geq 1, X1=1, Y1=1, \texttt{new2(X1, Y1, N)}.
  \item \texttt{new2(X, Y, N)} :- X=1, Y=1, \texttt{N}\geq 1, X1=2, Y1=3, \texttt{new3(X1, Y1, N)}.
  \item \texttt{new3(X, Y, N)} :- X1\geq 1, Y1\geq X1, X<N, X1=X+1, Y1=X1+Y, \texttt{new3(X1, Y1, N)}.
  \item \texttt{new3(X, Y, N)} :- Y\geq 1, \texttt{N}\geq 1, X\geq N, Y<X.
  \item \texttt{false} :- \texttt{N}\geq 1, Y\geq 1, X\geq N, Y<X, \texttt{new3(X, Y, N)}.
\end{itemize}
Iterated CHC Specialization: *SumUpto* Example

\[
\begin{align*}
\text{false} & : - N>=1, X=0, Y=0, p(X, Y, N). \\
p(X, Y, N) & : - X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N). \\
p(X, Y, N) & : - X>=N, Y<X. \\
\end{align*}
\]

**VC\(_0\)**

\[
\begin{align*}
\text{false} & : - N>=1, X1=1, Y1=1, \text{new2}(X1, Y1, N). \\
\text{new2}(X, Y, N) & : - X=1, Y=1, N>1, X1=2, Y1=3, \text{new3}(X1, Y1, N). \\
\text{new3}(X, Y, N) & : - X1>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, \text{new3}(X1, Y1, N). \\
\text{new3}(X, Y, N) & : - Y>=1, N>=1, X>=N, Y<X. \\
\end{align*}
\]

**VC\(_1\)**

\[
\begin{align*}
\text{false} & : - N>=1, X=0, Y=0, p(X, Y, N). \\
p(X, Y, N) & : - X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N). \\
p(X, Y, N) & : - X>=N, Y<X. \\
\text{false} & : - N>=1, X1=1, Y1=1, \text{new2}(X1, Y1, N). \\
\text{new2}(X, Y, N) & : - X=1, Y=1, N>1, X1=2, Y1=3, \text{new3}(X1, Y1, N). \\
\text{new3}(X, Y, N) & : - X1>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, \text{new3}(X1, Y1, N). \\
\text{new3}(X, Y, N) & : - Y>=1, N>=1, X>=N, Y<X. \\
\end{align*}
\]

**VC\(_2\)**

\[
\begin{align*}
\text{false} & : - N>=1, X=0, Y=0, p(X, Y, N). \\
p(X, Y, N) & : - X<N, X1=X+1, Y1=Y+2, p(X1, Y1, N). \\
p(X, Y, N) & : - X>=N, Y<X. \\
\text{false} & : - N>=1, X1=1, Y1=1. \\
\text{new2}(X, Y, N) & : - X=1, Y=1, N>1, X1=2, Y1=3, \text{new2}(X, Y, N). \\
\text{new3}(X1, Y1, N) & : - X1>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, \text{new3}(X, Y, N). \\
\text{false} & : - N>=1, Y>=1, X>=N, Y<X, \text{new3}(X, Y, N). \\
\end{align*}
\]

**VC\(_3\)**

\[
\begin{align*}
\text{false} & : - N>=1, Y>=1, X>=N, Y<X, \text{new4}(X, Y, N). \\
\end{align*}
\]

No constrained facts. VC\(_3\) is satisfiable.
VeriMAP architecture
Short demo
Experimental evaluation

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

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

<table>
<thead>
<tr>
<th></th>
<th>VeriMAP ($Gen_{ph}$)</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>159</td>
<td>91</td>
</tr>
<tr>
<td>2 safe problems</td>
<td>154</td>
<td>112</td>
<td>137</td>
<td>74</td>
</tr>
<tr>
<td>3 unsafe problems</td>
<td>31</td>
<td>26</td>
<td>22</td>
<td>17</td>
</tr>
<tr>
<td>4 incorrect answers</td>
<td>0</td>
<td>9</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>5 false alarms</td>
<td>0</td>
<td>8</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>6 missed bugs</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>7 errors</td>
<td>0</td>
<td>18</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>8 timed-out problems</td>
<td>31</td>
<td>51</td>
<td>52</td>
<td>92</td>
</tr>
<tr>
<td>9 total score</td>
<td>339 (0)</td>
<td>210 (-40)</td>
<td>268 (-28)</td>
<td>113 (-52)</td>
</tr>
<tr>
<td>10 total time</td>
<td>107717.34</td>
<td>15788.21</td>
<td>15770.33</td>
<td>27757.46</td>
</tr>
<tr>
<td>11 average time</td>
<td>57.93</td>
<td>114.41</td>
<td>99.18</td>
<td>305.03</td>
</tr>
</tbody>
</table>

Table 1: Verification results using VeriMAP, ARMC, HSF(C) and TRACER. For each column the sum of the values of lines 1, 4, 7, and 8 is 216, which is the total number of the verification problems we have considered. The timeout limit is five minutes. Times are in seconds.
Array constraints

- if $a[i] = v$ then $\text{read}(A, I, V)$ holds
- if $a[i] := v$ then $\text{write}(A, I, V, B)$ holds, that is $B$ is an array identical to $A$ except that $B$ has value $V$ in position $I$

- Constraint Handling Rules [Frühwirth et al.] for constraint reasoning

  **Array-Congruence-1:** if $i=j$ then $a[i]=a[j]$
  \[
  \text{read}(A, I, X) \setminus \text{read}(A_1, J, Y) \iff A = A_1, I = J \mid X = Y.
  \]

  **Array-Congruence-2:** if $a[i] \neq a[j]$ then $i \neq j$
  \[
  \text{read}(A, I, X), \text{read}(A_1, J, Y) \Rightarrow A = A_1, X \neq Y \mid I \neq J.
  \]

  **Read-Over-Write:** if $i=j$ then $x=y$
  \[
  \text{write}(A, I, X, A_1) \setminus \text{read}(A_2, J, Y) \iff A_1 = A_2 \mid (I = J, X = Y) ; (I \neq J, \text{read}(A, J, Y)).
  \]
Array constraint generalization

- Logic variables are decorated with identifiers of the imperative program

\[
\text{new3}(I,N,A) :- E+1=F, E \geq 0, I>F, G \geq H, N>F, N \leq I+1, \\
\text{read}(A,E^j,G^{a[j]}), \text{read}(A,F^{j_1},H^{a[j_1]}), \text{reach}(I,N,A).
\]

\[
\text{new4}(I,N,A) :- E+1=F, E \geq 0, I>F, G \geq H, I=1+I_1, I_1+2 \leq C, N \leq I_1+3, \\
\text{read}(A,E^j,G^{a[j]}), \text{read}(A,F^{j_1},H^{a[j_1]}), \text{read}(A,P^i,Q^{a[i]}), \\
\text{reach}(I,N,A).
\]

\[
\text{new5}(I,N,A) :- E+1=F, E \geq 0, I>F, G \geq H, N>F, \\
\text{read}(A,E^j,G^{a[j]}), \text{read}(A,F^{j_1},H^{a[j_1]}), \text{reach}(I,N,A).
\]
Experimental evaluation

Table 1. Verification results using VeriMAP and Z3 on a set of 88 verification problems: the verification precision (that is, the number of solved problems) and the average time. Times are in seconds.

<table>
<thead>
<tr>
<th>(1) $G = VCGen$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>average time</td>
<td>0.1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(2) $GZ = VCGen ; Z3$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>verification precision</td>
<td>49</td>
</tr>
<tr>
<td>average time</td>
<td>3.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(3) $GT = VCGen ; VCTransf$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen function parameters</td>
<td>$H$, $I$, $\cap$</td>
</tr>
<tr>
<td>verification precision</td>
<td>60</td>
</tr>
<tr>
<td>average time</td>
<td>7.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(4) $GTZ = VCGen ; VCTransf ; Z3$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Gen function parameters</td>
<td>$H$, $I$, $\cap$</td>
</tr>
<tr>
<td>verification precision</td>
<td>67</td>
</tr>
<tr>
<td>average time</td>
<td>16.8</td>
</tr>
</tbody>
</table>

References

- [DFPP – Fundamenta Informaticae 2017]
- [http://map.uniroma2.it/smc/array-chr/]

Frankfurt am Main, September 5th, 2018
Verification of relational properties
Relational Properties

• Stepwise program development

Optimization
Refactoring
New features

• Proving relations between fragments of program versions (e.g., equivalence) may be easier than proving the correctness of the new version from scratch.
• ... proving relations between executions of the same program with different input
An Example

```c
void sum_upto() {
    z1=f(x1);
}
int f(int n1){
    int r1;
    if (n1 <= 0) {
        r1 = 0;
    } else {
        r1 = f(n1 - 1) + n1;
    }
    return r1;
}

z1 = \sum_{n1=0}^{x1} n1 = x1*(x1+1)/2

(Non-tail) recursive

void prod() {
    z2 = g(x2,y2);
}
int g(int n2, int m2){
    int r2;
    r2=0;
    while (n2 > 0) {
        r2 += m2;
        n2--;
    }
    return r2;
}

z2 = x2 * y2

Iterative

• Relational property
  if  x1=x2 and x2≤y2 before execution of sum_upto and prod
  and execution terminates, then  z1≤z2
Verification of Relational Properties

- State-of-the-art verification methods for relational properties are specific for the given programming language PL and class of properties RL [Benton 2004, Barthe et al. 2011, Felsing et al. 2014]

P1, P2: programs in programming language PL
rel: property in logic RL

Verifier for PL and RL

- true
- false
- unable to verify
Verification through Horn Clause Transformation

CHC as a meta-language for programs, properties, and semantics.

P1 rel P2

Translator to CHC

Transformer of CHC

CHC Solver (Eldarica, Z3, ...)

Parametric w.r.t. PL and RL.
Relational properties

- **Terminating computation**

  \[ \langle P, env_0 \rangle \downarrow env_h \text{ iff } \langle l_0 : c_0, env_0 \rangle \Rightarrow^* \langle l_h : \text{halt}, env_h \rangle \]

- **Relational Property** P1, P2 programs with disjoint variables, \( \varphi, \psi \) constraints

  \[ \{ \varphi \} P1 \sim P2 \{ \psi \} \]

  is valid iff for all disjoint environments \( env_{01} \) and \( env_{02} \)

  if \( \models \varphi[env_{01} \cup env_{02}] \), \( \langle P1, env_{01} \rangle \downarrow env_{h1} \), \( \langle P2, env_{02} \rangle \downarrow env_{h2} \)

  then \( \models \psi[env_{h1} \cup env_{h2}] \)
Example, cont’d

```
void sum_upto() {
    z1=f(x1);
}
int f(int n1){
    int r1;
    if (n1 <= 0) {
        r1 = 0;
    } else {
        r1 = f(n1 - 1) + n1;
    }
    return r1;
}

    \[ z1 = \sum_{n1=0}^{x1} n1 = x1\times(x1+1)/2 \]

(Non-tail) recursive

void prod() {
    z2 = g(x2,y2);
}
int g(int n2, int m2){
    int r2;
    r2=0;
    while (n2 > 0) {
        r2 += m2;
        n2--;
    }
    return r2;
}

    \[ z2 = x2 \times y2 \]

Iterative

Relational Property:
{\( x1=x2 \land x2\leq y2 \)} sum_upto \sim prod \{ z1 \leq z2 \}
Encoding the Transition Semantics in CHCs

- **Reflexive-transitive closure** $\Rightarrow^*$:
  
  \[
  \text{reach}(C,C) \leftarrow \\
  \text{reach}(C,C2) \leftarrow \text{tr}(C,C1), \text{reach}(C1,C2)
  \]

- **Terminating computation** $\langle P, \text{env}_0 \rangle \Downarrow \text{env}_n$ [input/output relation of $P$]:

  \[
  p(X,X') \leftarrow \text{initConf}(C,X), \text{reach}(C,C'), \text{finalConf}(C',X')
  \]

  - $\text{initConf}(C,X)$: $X$ is the value of the variables in the initial configuration $C$
  - $\text{finalConf}(C',X')$: $X'$ is the value of the variables in the final configuration $C'$
Translating Relational Properties into CHCs

• \{\varphi\} P_1 \sim P_2 \{\psi\}

\[ Prop: \quad \text{false} \leftarrow \text{pre}(X,Y), p_1(X,X'), p_2(Y,Y'), \text{neg\_post}(X',Y') \]

\[ \varphi \quad \text{P}_1 \quad \text{P}_2 \quad \neg\psi \]

\( X,Y,X',Y' \): tuples of values for the variables of \( P_1, P_2 \), resp.

• \( T_{prop} = \{Prop\} \cup \{\text{clauses for } p_1 \text{ and } p_2\} \)

Correctness of Translation:

\[ \{\varphi\} P_1 \sim P_2 \{\psi\} \text{ is valid iff } T_{prop} \text{ is satisfiable} \]

• Example:  \( \text{false} \leftarrow \text{X}_1=\text{X}_2, \text{X}_2\leq\text{Y}_2, \text{Z}_1'>\text{Z}_2', \)

\[ \text{sum\_upto}(\text{X}_1,\text{Z}_1,\text{X}_1',\text{Z}_1'), \text{prod}(\text{X}_2,\text{Y}_2,\text{Z}_2,\text{X}_2',\text{Y}_2',\text{Z}_2') \]
Example Cont’d: CHC Specialization

\[
\text{false} \leftarrow X_1 = X_2, X_2 \leq Y_2, Z_1' > Z_2', \\
\text{sum}_\text{upto}(X_1, Z_1, X_1', Z_1'), \text{prod}(X_2, Y_2, Z_2, X_2', Y_2', Z_2')
\]

\[
\text{false} \leftarrow X_1 = X_2, X_2 \leq Y_2, Z_1' > Z_2', \text{su}(X_1, Z_1'), \text{pr}(X_2, Y_2, Z_2')
\]

\[
\text{su}(X, Z) \leftarrow f(X, Z)
\]

\[
f(N, Z) \leftarrow N \leq 0, Z = 0
\]

\[
f(N, Z) \leftarrow N_1, N_1 = N - 1, Z = R + N, f(N_1, R)
\]

\[
\text{pr}(X, Y, Z) \leftarrow W = 0, g(X, Y, W, Z)
\]

\[
g(N, P, R, R) \leftarrow N \leq 0
\]

\[
g(N, P, R, R_2) \leftarrow N_1, N_1 = N - 1, R_1 = P + R, g(N_1, P, R_1, R_2)
\]

+ clauses for sum_upto and prod

Specialized predicates

CHC Specializer

sum_upto(X1,Z1,X1’,Z1’), prod(X2,Y2,Z2,X2’,Y2’,Z2’)

false \(\leftarrow\) \(X_1 = X_2, X_2 \leq Y_2, Z_1' > Z_2',\)

\(\text{sum}_\text{upto}(X_1, Z_1, X_1', Z_1'), \text{prod}(X_2, Y_2, Z_2, X_2', Y_2', Z_2')\)
Limitations of the Specialized CHCs

• To show the satisfiability of

\[ \text{false} \leftarrow c(X,Y), \ p_1(X), \ p_2(Y) \]

a CHC solver looks for \( c_1(X), \ c_2(Y) \) such that in \( T_{SP} \cup \text{Th} \):

\[
\begin{align*}
p_1(X) & \rightarrow c_1(X) \\
p_2(Y) & \rightarrow c_2(Y) \\
c_1(X), \ c_2(Y), \ c(X,Y) & \rightarrow \text{false}
\end{align*}
\]

• To show the satisfiability of

\[ \text{false} \leftarrow X_1=X_2, \ X_2 \leq Y_2, \ Z_1'>Z_2', \ su(X_1,Z_1'), \ pr(X_2,Y_2,Z_2') \]

a CHC solver has to show that:

\[
\begin{align*}
su(X_1,Z_1') & \rightarrow Z_1' \leq 1 + \ldots + X_1 \\
pr(X_2,Y_2,Z_2') & \rightarrow Z_2' \geq X_2 \cdot Y_2 \\
Z_1' \leq 1 + \ldots + X_1, \ Z_2' & \geq X_2 \cdot Y_2, \ X_1=X_2, \ X_2 \leq Y_2, \ Z_1'>Z_2' & \rightarrow \text{false}
\end{align*}
\]

• Impossible for CHC solvers over LIA!
Nonlinear constraints cannot be derived.
Example Cont’d: Predicate Pairing

false ← X1=X2, X2≤Y2, Z1’>Z2’,
   su(X1,Z1’), pr(X2,Y2,Z2’)

su(X,Z) ← f(X,Z)

f(N,Z) ← N≤ 0, Z=0

f(N,Z) ← N1, N1=N−1, Z=R+N, f(N1,R)

pr(X,Y,Z) ← W=0, g(X,Y,W,Z)

g(N,P,R,R) ← N≤ 0

g(N,P,R,R2) ←
   N1, N1=N−1, R1=P+R,
   g(N1,P,R1,R2)

•  fg(N,Z1’,Y,0,Z2’) → N>Y v Z1’≤ Z2’
   (N>Y v Z1’≤ Z2’) ∧ N≤ Y ∧ W=0 ∧ Z1’>Z2’ → false

• Non-linear arithmetic relations not needed for proving satisfiability.
   CHC solvers over LIA (Eldarica, Z3) can prove satisfiability.
Inferring Inter-Predicate Relations via Predicate Pairing

• Introduce new predicates standing for conjunctions:

false ← c(X,Y), \( p1(X), p2(Y) \)

• Predicate pairing derives new clauses for conjunctions of predicates by unfold/fold transformations and preserves satisfiability.

• To prove satisfiability find constraint \( d(X,Y) \) such that:

\[
\begin{align*}
p12(X,Y) &\rightarrow d(X,Y) \\
d(X,Y), c(X,Y) &\rightarrow false
\end{align*}
\]

• \( d(X,Y) \) captures relations between the variables of \( p1 \) and the variables of \( p2 \).
Properties of the CHC transformation rules

• CHC transformation rules preserve satisfiability
  [Tamaki-Sato 84, Etalle-Gabbrielli 96]

• Theorem [DFPP 17]
  Let $A$ be a subset of the constraints of Th.
  Let $P \rightarrow \ldots \rightarrow Q$ be a transformation sequence
  if $P$ has an $A$-definable model then $Q$ has an $A$-definable model

• Thus, CHC transformation rules preserve solvability (in abstract domains too).
  Example: constraints over LIA.
  $A$ can be LIA or Octagons, difference constraints, ....
Implementation in VeriMAP

C program 1 \rightarrow (T) Translator into CHCs (VERIMAP)
C program 2 \rightarrow relational property

(1) Predicate Pairing (VERIMAP)
(2) Translator into SMT-LIB (VERIMAP)

SMT Solver \rightarrow true, false, unknown
Short demo
Verification Problems

Types of Verified Properties and Programs

• **NLIN**: nonlinear or nested recursion
  (e.g. some Ackermann variants, Sudan, McCarthy’s 91, Dijkstra’s *fusc*

• **MON**: monotonicity
  \[
  \text{if } i_1 \geq i_2 \text{ then } o_1 \geq o_2
  \]

• **INJ**: injectivity
  \[
  \text{if } i_1 \neq i_2 \text{ then } o_1 \neq o_2
  \]

• **FUN**: functional dependency among variables
  \[
  \text{if } i_1 = i_2 \text{ then } o_1 = o_2
  \]

• **NINT**: non-interference
  public output variables depend on public input variables only

• **LOPT**: loop and other compiler optimizations
  e.g. loop-unswitching, loop-fission, loop-fusion, loop-reversal, strength-reduction
Verification Problems

Types of Verified Properties and Programs

• **ITE**: equivalence of two iterative programs on integers
• **ARR**: equivalence of two programs on arrays
• **REC**: equivalence of two recursive programs
• **I-R**: equivalence of an iterative and a (non-tail) recursive program
  e.g. greatest common divisor, n-th triangular number
• **COMP**: composition of different number of loops of integer and array progr.
• **PCOR**: partial correctness properties of an iterative program
  wrt a recursive functional postcondition

31 programs out of 163 are encoded using non-linear CHC
Experimental evaluation

- Timeout: 300 seconds
- No timeout occurred during the application of the PP strategy.
- CHC size increase due to PP but no performance degradation

<table>
<thead>
<tr>
<th>Problems</th>
<th>Z3 before PP</th>
<th>PP</th>
<th>Z3 after PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category</td>
<td>P</td>
<td>$S_1$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>(1) NLIN</td>
<td>13</td>
<td>4</td>
<td>16.11</td>
</tr>
<tr>
<td>(2) MON</td>
<td>18</td>
<td>1</td>
<td>1.04</td>
</tr>
<tr>
<td>(3) INJ</td>
<td>11</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>(4) FUN</td>
<td>7</td>
<td>4</td>
<td>1.39</td>
</tr>
<tr>
<td>(5) NINT</td>
<td>18</td>
<td>3</td>
<td>0.27</td>
</tr>
<tr>
<td>(6) LOPT</td>
<td>20</td>
<td>2</td>
<td>4.83</td>
</tr>
<tr>
<td>(7) ITE</td>
<td>22</td>
<td>5</td>
<td>26.67</td>
</tr>
<tr>
<td>(8) ARR</td>
<td>6</td>
<td>1</td>
<td>7.45</td>
</tr>
<tr>
<td>(9) REC</td>
<td>15</td>
<td>6</td>
<td>2.89</td>
</tr>
<tr>
<td>(10) I-R</td>
<td>4</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>(11) COMP</td>
<td>10</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>(12) PCOR</td>
<td>19</td>
<td>5</td>
<td>83.93</td>
</tr>
<tr>
<td><strong>Total number</strong></td>
<td>163</td>
<td>31</td>
<td>144.58</td>
</tr>
<tr>
<td><strong>Average Time</strong></td>
<td>4.66</td>
<td>0.81</td>
<td>0.90</td>
</tr>
</tbody>
</table>
Comments

• Our method for relational verification:
  Translation to CHCs;
  Satisfiability-Preserving Transformations of CHCs;
  CHC Solving
• Parametric wrt programming language
• Fully automatic and effective on small-sized programs

Future work
• Proving relations across programming languages to validate program translation/compilation

References
• [DFPP – SAS 16]  [DFPP – TPLP 17]
• http://map.uniroma2.it/relprop/
Verification of programs with inductively-defined data structures
Verification of functional programs

- **OCaml**: A statically typed, functional, higher-order, OO language
- Computing the **sum** and the **maximum** of the absolute values of the elements of a list:

  ```ocaml
type list = Nil | Cons of int * list

let rec listsum l = match l with
  | Nil -> 0
  | Cons(x, xs) → (abs x) + listsum xs

let rec listmax l = match l with
  | Nil -> 0
  | Cons(x, xs) → let m = listmax xs in max (abs x) m

- (Relational) **Property**: \( \forall l. \text{listsum}(l) \geq \text{listmax}(l) \)
Translation into CHCs

- The OCaml program is translated into CHCs:

\[
\begin{align*}
\text{listsum}([], S) & \leftarrow S=0 \\
\text{listsum}([X \mid Xs], S) & \leftarrow S=S1+A, \\text{abs}(X, A), \\text{listsum}(Xs, S1) \\
\text{listmax}([], M) & \leftarrow M=0 \\
\text{listmax}([X \mid Xs], M) & \leftarrow \text{abs}(X, A), \text{max}(A, M1, M), \\text{listmax}(Xs, M1) \\
\text{abs}(X, A) & \leftarrow (X \geq 0, A=X) \lor (X<0, A=-X) \\
\text{max}(A, M1, M) & \leftarrow (A \geq M1, M=A) \lor (A<M1, M=M1)
\end{align*}
\]

- The property is translated into a CHC query:

\[
\begin{align*}
\text{false} & \leftarrow S<M, \\text{sum}(L, S), \text{max}(L, M)
\end{align*}
\]

- The clauses are satisfiable but CHC solvers do not solve them because models are infinite formulas in the quantifier-free theory of integer lists:

\[
\begin{align*}
\text{listsum}(L, S) & \leftrightarrow (L=[], S=0) \lor (L=[X], \text{abs}(X, S)) \lor (L=[X,Y], \text{abs}(X, A), \text{abs}(Y, B), S=A+B) \lor \ldots \\
\text{listmax}(L, M) & \leftrightarrow (L=[], M=0) \lor (L=[X], \text{abs}(X, M)) \lor \ldots
\end{align*}
\]
Solving CHCs on inductively defined data types by induction

- Solution 1: Extending CHC solving with induction.
- Proof of satisfiability, by induction on list $L$:

  $\forall L,S,M. \text{listsum}(L,S), \text{listmax}(L,M) \rightarrow S \geq M$

  and hence $\text{listsum}(L,S), \text{listmax}(L,M), S < M \rightarrow \text{false}$

Solving CHCs on inductively defined data types by CHC transformation

• Solution 2 (this work): Transform CHCs on inductive data types into equisatisfiable CHCs without inductive data types (e.g., on integers or booleans):

\[
\text{list-sum-max}(S,M) \leftarrow S=0, M=0 \\
\text{list-sum-max}(S,M) \leftarrow S=S1+A, \text{abs}(X,A), \text{max}(A,M1,M), \text{list-sum-max}(S1,M1) \\
\text{false} \leftarrow S<M, \text{list-sum-max}(S,M)
\]

• Solved by Z3, without induction.

Solution: \( \text{list-sum-max}(S,M) \rightarrow S \geq M, M \geq 0 \)

• No infinite models are needed to show satisfiability
Eliminating inductive data structures

- Transformations for eliminating inductive data structures: Deforestation [Wadler ‘88], Unnecessary Variable Elimination by Unfold/Fold [PP ‘91], Conjunctive Partial Deduction [De Schreye et al. ‘99]

- **Define** a new predicate:
  \[
  \text{list-sum-max}(S, M) \leftarrow \text{listsum}(L, S), \text{listmax}(L, M)
  \]

- **Unfold**:
  \[
  \begin{align*}
  \text{list-sum-max}(S, M) \leftarrow & \ S=0, \ M=0 \\
  \text{list-sum-max}(S, M) \leftarrow & \ S=S1+A, \ \text{abs}(X, A), \ \text{max}(A, M1, M), \\
  & \ \text{listsum}(Xs, S1), \ \text{listmax}(Xs, M1)
  \end{align*}
  \]

- **Fold** (eliminate lists):
  \[
  \begin{align*}
  \text{list-sum-max}(S, M) \leftarrow & \ S=0, \ M=0 \\
  \text{list-sum-max}(S, M) \leftarrow & \ S=S1+A, \ \text{abs}(X, A), \ \text{max}(A, M1, M), \\
  & \ \text{list-sum-max}(S1, M1)
  \end{align*}
  \]

  \[
  \text{false} \leftarrow S<M, \ \text{list-sum-max}(S, M)
  \]
The Elimination Algorithm EC

1. Define new predicate(s) with Ind. Data Structs in the body only
2. Unfold new predicate(s)
3. Use Functionality (if possible)
4. Fold to eliminate Ind. Data Structs

Ind. Data Structs?

- yes
- no
Termination

- Algorithm $E$ terminates if
  - the query has no sharing cycles
  - the other clauses have a disjoint, quasi-descending slice decomposition

\[
\begin{align*}
\text{min}(X, Y, Z) & \leftarrow X < Y, \ Z = X \\
\text{min}(X, Y, Z) & \leftarrow X \geq Y, \ Z = Y \\
\text{min\_leaf}(\text{leaf}, M) & \leftarrow M = 0 \\
\text{min\_leaf}(\text{node}(X, L, R), M) & \leftarrow M = M3 + 1, \ \text{min\_leaf}(L, M1), \ \text{min\_leaf}(R, M2), \ \\
& \quad \text{min}(M1, M2, M3) \\
\text{left\_drop}(N, \text{leaf}, \text{leaf}) & \leftarrow \\
\text{left\_drop}(N, \text{node}(X, L, R), \text{node}(X, L, R)) & \leftarrow N \leq 0 \\
\text{left\_drop}(N, \text{node}(X, L, R), T) & \leftarrow N \geq 1, \ N1 = N - 1, \ \text{left\_drop}(N1, L, T) \\
\text{false} & \leftarrow N \geq 0, \ M + N < K, \ \text{left\_drop}(N, T, U), \ \text{min\_leaf}(U, M), \ \text{min\_leaf}(T, K)
\end{align*}
\]
A nonterminating transformation

- A property of lists

\[
\text{if } M = N \text{ then } A = Xs
\]

\[
\begin{align*}
\text{append}([ ], Ys, Ys) & \leftarrow \\
\text{append}([X|Xs], Ys, [Z|Zs]) & \leftarrow X = Z, \\
\text{append}(Xs, Ys, Zs) & \\
\text{drop}(N, [ ], [ ]) & \leftarrow \\
\text{drop}(N, [X|Xs], [Y|Xs]) & \leftarrow N = 0, X = Y \\
\text{drop}(N, [X|Xs], Ys) & \leftarrow N \neq 0, N1 = N - 1, \\
\text{drop}(N1, Xs, Ys) & \\
\text{false} & \leftarrow M = N, \text{ take}(M, Xs, Ys), \text{ drop}(N, Xs, Zs), \text{ append}(Ys, Zs, A), \text{ diff_list}(A, Xs)
\end{align*}
\]
Verification of OCaml Programs

1. OCaml Program

2. Translation to CHCs \([\text{RCaml, Unno \& al. 2017}]\)

3. Algorithm \(\text{EC} [\text{VeriMAP, De Angelis \& al. 2014-18}]\)

4. CHCs w/o Ind. Data S

5. Z3 CHC solver with \text{SPACER} engine \([\text{Komuravelli \& al. 2013}]\)

6. sat/unsat/unknown
Experimental evaluation

- Benchmark:
  - 70 OCaml small (but non-trivial) programs on lists/trees from RCaml and IsaPlanner (a proof planner for ISABELLE)
  - 35 more OCaml programs (e.g., binary search trees)

<table>
<thead>
<tr>
<th>Problem Set</th>
<th>$n$</th>
<th>$S_{Z3}$</th>
<th>$T_{Z3}$</th>
<th>$S_{EC;Z3}$</th>
<th>$T_{EC;Z3}$</th>
<th>$S_{RCAML}$</th>
<th>$T_{RCAML}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FirstOrder</td>
<td>57</td>
<td>3</td>
<td>0.09</td>
<td>47</td>
<td>37.64</td>
<td>41</td>
<td>216.59</td>
</tr>
<tr>
<td>HigherOrderInstances</td>
<td>13</td>
<td>1</td>
<td>0.04</td>
<td>11</td>
<td>8.33</td>
<td>10</td>
<td>45.40</td>
</tr>
<tr>
<td>MoreLists</td>
<td>16</td>
<td>3</td>
<td>13.87</td>
<td>14</td>
<td>11.27</td>
<td>10</td>
<td>119.01</td>
</tr>
<tr>
<td>MoreTrees</td>
<td>19</td>
<td>5</td>
<td>20.18</td>
<td>19</td>
<td>26.79</td>
<td>5</td>
<td>55.16</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>105</td>
<td>12</td>
<td>34.18</td>
<td>91</td>
<td>84.03</td>
<td>66</td>
<td>436.17</td>
</tr>
<tr>
<td><strong>Avg time</strong></td>
<td></td>
<td></td>
<td></td>
<td>2.85</td>
<td>0.92</td>
<td></td>
<td>6.61</td>
</tr>
</tbody>
</table>
Comments

- Transformation is a viable alternative to induction to solve CHCs on data structures
- We presented transformation algorithms which are effective on small, non-trivial examples

Future work
- Higher-order functional programs
- Discover and apply lemmata to eliminate inductive data structures

References
- [DFPP - TPLP 18]
- https://fmlab.unich.it/iclp2018/
Verification of time-aware business processes
**Business Processes**

- **Business processes** are ‘graphs’ for coordinating the activities of an organization towards a business goal.

- **An example: Purchase Order.** A customer adds items to the shopping cart and pays. Then, the vendor issues and sends the invoice, and in parallel, prepares and delivers the order.

There is no information on the durations of tasks.
Time-Aware Business Processes

- **Information on the duration**: Intervals: $d \in [d_{\text{min}}, d_{\text{max}}] \subset \mathbb{N}$

Two problems:
- **Time-Reachability**: checking whether or not to go from $s$ to $e$ takes less than $k$ units of time.
- **Controllability**: finding the durations of some controllable tasks so that a given time-reachability property holds.
Business Process Modeling and Notation (BPMN)

*Graphical notation* for modeling organizational processes. BPMN is a standard.

- **Tasks**: atomic activities
- **Events**: something that happens
- **Gateways**: either branching or merging
- **Flows**: order of execution (drawn as *arrows*)
Branch Gateways

- single incoming flow, multiple outgoing flows

- **exclusive** branch gateway (XOR)
  - upon activation of the incoming flow *exactly one* outgoing flow is activated

- **parallel** branch gateway (AND)
  - upon activation of the incoming flow *all* outgoing flows are activated
Branch Gateways

- single incoming flow, multiple outgoing flows

- **exclusive** branch gateway (XOR)
  - upon activation of the incoming flow *exactly one* outgoing flow is activated

- **parallel** branch gateway (AND)
  - upon activation of the incoming flow *all* outgoing flows are activated
Merge Gateways

- multiple incoming flows, single outgoing flow
  - **exclusive** merge gateway (XOR)
    - the outgoing flow is activated upon activation of *one* of the incoming flows
  - **parallel** merge gateway (AND)
    - the outgoing flow is activated upon activation of *all* the incoming flows
Merge Gateways

- multiple incoming flows, single outgoing flow

- **exclusive merge gateway** (XOR)
  - the outgoing flow is activated upon activation of *one* of the incoming flows

- **parallel merge gateway** (AND)
  - the outgoing flow is activated upon activation of *all* the incoming flows
Semantics of time-aware BPMN

• Transition relation between states: \( <F,t> \rightarrow <F',t'> \)

• \( F \): a set of *fluents* (i.e., a set of properties that hold at time point \( t \))
  - *begins*\((x)\) \( x \) begins its execution (enactment)
  - *enacting*\((x,r)\) \( x \) is executing with \( r \) residual time to completion
  - *completes*\((x)\) \( x \) completes its execution
  - *enables*\((x,y)\) \( x \) enables its successor \( y \)
    \( x, y \) denote either tasks, or events, or gateways

• *seq*\((x,y)\) there is an arrow from \( x \) to \( y \)

• \( t \): time point (i.e., a non-negative integer)

  *duration*\((x,d)\) the duration of \( x \) is \( d \)
Semantics of time-aware BPMN

\[ \text{task}(x) \leftarrow \]
\[ \text{duration}(x, d) \leftarrow 3 \leq d \leq 4 \]

\[ \text{enacting}(x, r) \text{ with } 0 \leq r \leq d \]
\[ \text{begins}(x) \quad \text{completes}(x) \]

- durations of events and gateways are assumed to be 0
Semantics of time-aware BPMN

Instantaneous transition:

\[ \text{begins}(x) \quad \rightarrow \quad \text{enacting}(x, d) \]

\[
(S_1) \quad \frac{\text{begins}(x) \in F \quad \text{duration}(x, d)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{\text{begins}(x)\}) \cup \{\text{enacting}(x, d)\}, t \rangle}
\]
Semantics of time-aware BPMN

Instantaneous transitions:

\[ (S_2) \quad \frac{\text{completes}(x) \in F \quad \text{par\_branch}(x)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{\text{completes}(x)\}) \cup \{\text{enables}(x, s) \mid \text{seq}(x, s)\}, t \rangle} \]

\[ (S_3) \quad \frac{\text{completes}(x) \in F \quad \text{not\_par\_branch}(x) \quad \text{seq}(x, s)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{\text{completes}(x)\}) \cup \{\text{enables}(x, s)\}, t \rangle} \]

\( S_2 \) If the parallel branch \( x \) completes,
then all its successors \( S \) are enabled, instantaneously

\[ < F, t > \rightarrow < F', t > \]
Semantics of time-aware BPMN

Instantaneous transitions:

\[(S_2)\]
\[
\frac{completes(x) \in F \quad par\_branch(x)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{completes(x)\}) \cup \{enables(x, s) \mid seq(x, s)\}, t \rangle}
\]

\[(S_3)\]
\[
\frac{completes(x) \in F \quad not\_par\_branch(x) \quad seq(x, s)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{completes(x)\}) \cup \{enables(x, s)\}, t \rangle}
\]

\[(S_2)\] If the parallel branch \(x\) completes,
then all its successors \(S\) are enabled, instantaneously
Semantics of time-aware BPMN

The time-elapsing transition:

\[
(S_7) \quad \frac{\text{no\_other\_premises}(F) \quad \exists x \exists r \text{ enacting}(x, r) \in F \quad m > 0}{\langle F, t \rangle \rightarrow \langle F \ominus m \setminus \text{Enbls}, t + m \rangle}
\]

where: (i) \(\text{no\_other\_premises}(F)\) holds iff none of the premises of rules \(S_1-\text{S}_6\) holds, (ii) \(m = \min\{r \mid \text{enacting}(x, r) \in F\}\), (iii) \(F \ominus m\) is the set \(F\) of fluents where every \(\text{enacting}(x, r)\) is replaced by \(\text{enacting}(x, r-m)\), and (iv) \(\text{Enbls} = \{\text{enables}(p, s) \mid \text{enables}(p, s) \in F\}\).

Time elapses when no instantaneous transition can occur.

All enacting tasks proceed in parallel for a time equal to the minimum of all residual times.
Weak Controllability

- Assume:
  - some tasks are \textit{controllable} (e.g., internal to the organization)
  - some tasks are \textit{uncontrollable} (e.g., external to the organization)

- \textbf{Weak Controllability:} For all durations of the uncontrollable tasks (within the given time intervals), we can determine durations of the controllable tasks (within the given time intervals), s.t. a state can be reached and a given time constraint is satisfied.

### Constraint:
\[ 3 \leq T_{\text{total}} \leq 7 \]

### A solution:
\[
\text{if } D_{\text{pur}} = 1 \text{ then } D_{\text{cc}} = D_{\text{col}} = 2 \text{ else } D_{\text{cc}} = D_{\text{col}} = 1
\]
Strong Controllability

Weak Controllability may not be useful when some uncontrollable tasks occur after controllable ones.

- **Strong Controllability**: We can determine durations of the controllable tasks (within the given time intervals) s.t., for all durations of the uncontrollable tasks (within the given time intervals), a state can be reached and a given time constraint is satisfied.
- The exact duration of the delivery is not known when packaging.

\[
\begin{align*}
&\text{constraint: } 4 \leq T_{\text{total}} \leq 7 \\
&a \text{ solution: } 1 \leq D_{\text{pack}} \leq 2
\end{align*}
\]
Instantaneous transition:

\[
\begin{align*}
\text{begins}(x) \quad \rightarrow \quad \text{enacting}(x, d)
\end{align*}
\]

\[
(S_1) \quad \frac{\text{begins}(x) \in F \quad \text{duration}(x, d)}{\langle F, t \rangle \rightarrow \langle (F \setminus \{\text{begins}(x)\}) \cup \{\text{enacting}(x, d)\}, t \rangle}
\]

\[\text{C1. } tr(s(F, T), s(FU, T), U, C) \leftarrow \text{select}(\{\text{begins}(X)\}, F), \text{ task_duration}(X, D, U, C), \text{ update}(F, \{\text{begins}(X)\}, \{\text{enacting}(X, D)\}, FU)\]

where \(U, C\) are tuples of \text{uncontrollable} and \text{controllable} durations, resp.
CHC interpreter of time-aware BPMN

C1. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}\{\text{begins}(X)\}, F), \text{task}\_\text{duration}(X, D, U, C), \\
\text{update}(F, \{\text{begins}(X)\}, \{\text{enacting}(X, D)\}, FU) \)

C2. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}\{\text{completes}(X)\}, F), \text{par}\_\text{branch}(X), \\
\text{findall}\{\text{enables}(X, S), (\text{seq}(X, S)), \text{Enbs}\}, \text{update}(F, \{\text{completes}(X)\}, \text{Enbs}, FU) \)

C3. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}\{\text{completes}(X)\}, F), \text{not}\_\text{par}\_\text{branch}(X), \text{seq}(X, S), \\
\text{update}(F, \{\text{completes}(X)\}, \{\text{enables}(X, S)\}, FU) \)

C4. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}(\text{Enbs}, F), \text{par}\_\text{merge}(X), \\
\text{findall}\{\text{enables}(P, X), (\text{seq}(P, X)), \text{Enbs}\}, \text{update}(F, \text{Enbs}, \{\text{begins}(X)\}, FU) \)

C5. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}\{\text{enables}(P, X)\}, F), \text{not}\_\text{par}\_\text{merge}(X), \\
\text{update}(F, \{\text{enables}(P, X)\}, \{\text{begins}(X)\}, FU) \)

C6. \( \text{tr}(s(F, T), s(FU, T), U, C) \leftarrow \text{select}\{\text{enacting}(X, R)\}, F), R = 0, \\
\text{update}(F, \{\text{enacting}(X, R)\}, \{\text{completes}(X)\}, FU) \)

C7. \( \text{tr}(s(F, T), s(FU, TU), U, C) \leftarrow \text{no}\_\text{other}\_\text{premises}(F), \text{member}(\text{enacting}(\_\_, \_\_), F), \\
\text{findall}(Y, (\_\_ = \text{enacting}(X, R), \text{member}(Y, F)), \text{Enacts}), \\
\text{mintime}(\text{Enacts}, M), M > 0, \text{decrease}\_\text{residual}\_\text{times}(\text{Enacts}, M, \text{EnactsU}), \\
\text{findall}(Z, (\_\_ = \text{enables}(P, S), \text{member}(Z, F)), \text{Enbs}), \\
\text{set}\_\text{union}(\text{Enacts}, \text{Enbs}, \text{EnactsEnbs}), \text{update}(F, \text{EnactsEnbs}, \text{EnactsU}, FU), \\
TU = T + M \)
reach: reflexive, transitive closure of the transition relation $tr$

$R1$: $reach(S,S,U,C) \leftarrow$

$R2$: $reach(S0,S2,U,C) \leftarrow tr(S0,S1,U,C), reach(S1,S2,U,C)$
Encoding Reachability

- **Reachability Property.**
  \[
  \text{RP : } \text{reachProp}(U,C) \leftarrow c(T,U,C), \text{ reach}(\text{init}, \text{fin}(T), U,C)
  \]
  where \( c(T,U,C) \) is a constraint

- **Initial state.** \( \text{init} : \ < \{\text{begins(}\text{start})\}\}, \ 0 \> \)

- **Final state.** \( \text{fin}(T) : \ < \{\text{completes(}\text{end})\}\}, \ T > \)
Let $Sem$ be the CHC encoding of semantics: $C1$-$C7$ (for $tr$) and $R1$-$R2$ (for $reach$).
Let $LIA$ be the theory of Linear Integer Arithmetics.

- **Weak Controllability**
  \[
  Sem \cup \{RP\} \cup LIA \models \forall U. \text{adm}(U) \rightarrow \exists C \text{ reachProp}(U,C)
  \]
  where $\text{adm}(U)$ iff the durations in $U$ belong to the given intervals

- **Strong Controllability**
  \[
  Sem \cup \{RP\} \cup LIA \models \exists C. \forall U. \text{adm}(U) \rightarrow \text{reachProp}(U,C)
  \]
Verifying controllability

- Validity of Weak and Strong Controllabilities:
  - cannot be proved by CHC solvers over LIA (e.g., Z3), because of the complex terms (such as those denoting sets) and the findall predicate in Sem
  - cannot be proved by CLP systems, because of $\exists \forall$ and $\forall \exists$
  - solvers and CLP systems have termination problems due to recursive reach.

- We developed special purpose algorithms for solving weak and strong controllability.

Reduce solving of $\exists \forall$ and $\forall \exists$ with recursive clauses to
- computing answers to queries
- solving a set of quantified LIA constraints
Experimental evaluation

Different tools have been used:

- **VeriMAP** for generating CHC
- **SICStus** Prolog: Computation of answer constraints
- **Z3**: SMT solver for checking quantified LIA formulas

Experimentation on various examples:

- Purchase order [DFMPP 2016]
- Request Day-Off Approval [Huai et al. 2010]
- STEMI: Emergency Department Admission [Combi et al. 2009]
- STEMI: Emergency Department + Coronary Care Unit Admission [Combi et al. 2012]
Comments

• Controllability was introduced in various contexts

• Future work
  - Larger fragment of BPMN: timers, interrupting events, ...
  - Data [Montali et al. 2013, Deutsch 2014, ...]
  - Ontologies for tasks, ...

• References
  - [DFMPP – LOPSTR 16] [DFMPP – RuleML+RR 17]
  - http://map.uniroma2.it/lopstr16/
Final comments

• We presented a flexible framework for CHC verification
  - parametric with respect to the semantics and the property
  - use of satisfiability-preserving and solvability-preserving CHC transformations
  - can improve precision state-of-the-art CHC solvers

• Future work
  - Make it more usable (better interface, web interface)
  - Make it more extensible (define API, hooks, ... )
  - Integrate external libraries and tools

• You are welcome to use it for your verification tasks.
  - We would be happy to help you!
Thank you
Encoding the Operational Semantics

function call \hspace{1cm} x=f(e_1,...,e_n);

“return” case

\[
\text{tr(} \text{cf(cmd(L,asgn(X,call(F,Es))), (D,S)), cf(cmd(L2,C2), (D2,S2)))} \\
\leftarrow \text{eval_list(Es,D,S,Vs), build_funenv(F,Vs,FEnv), firstlab(F,FL), at(FL,C), reach( cf(cmd(FL,C), (D,FEnv)), cf(cmd(LR,return(E)),(D1,S1))), eval(E,(D1,S1),V), update(((D1,S),X,V,(D2,S2)), nextlab(L,L2), at(L2,C2))}
\]

source configuration

target configuration

evaluate function parameters

build function environment

first label and command function def

function execution

return

evaluate returned expression

update caller environment

next label and command
VCs Multi-Step Semantics

false ← X>=1,Y>=1,X1=<-1, new3(X,Y, X1,Y1)
new3(X,Y, X1,Y1) ← X+1=<Y, new4(X,Y, X1,Y1)
new3(X,Y, X1,Y1) ← X>=Y+1, new4(X,Y, X1,Y1)
new3(X,Y, X,Y) ← X=Y
new4(X,Y, X3,Y3) ← X>=Y+1, A=X, B=Y, X2=R1,
new6(X,Y,A,B,R, X1,Y1,A1,B1,R1),
new3(X2,Y1, X3,Y3)
new4(X,Y, X3,Y3) ← X=<Y, A=Y, B=X, Y2=R1,
new6(X,Y,A,B,R, X1,Y1,A1,B1,R1),
new3(X1,Y2, X3,Y3)
new6(X,Y,A,B,R, X,Y,A,B,R1) ← R1=A-B

VCs generated by using the multi-step semantics

- Non linear recursive: multiple atoms in the body
- Predicate arity is even (variables for source and target configurations)
Small-Step Semantics

- Keep a stack of activation frames

- **Function call**: push an element on top of the stack
  
  \[
  \text{tr(cf(cmd(L, asgn(X, call(F, Es))), D, T),}
  \text{ cf(cmd(FL, C), D, [frame(L1, X, FEnv) | T]))} \leftarrow
  \text{nextlab(L, L1),}
  \text{ loc_env(T, S), eval_list(Es, D, S, Vs),}
  \text{ build_funenv(F, Vs, FEnv),}
  \text{ firstlab(F, FL), at(FL, C).}
  \]

  - **L1**: label where to jump after returning
  - **X**: value returned by the function call
  - **FEnv**: local environment used during the execution of the function call

- **Function return**: pop an element from the stack
  
  \[
  \text{tr(cf(cmd(L, return(E)), D, [frame(L1, X, S) | T]),}
  \text{ cf(cmd(L1, C), D1, T1))} \leftarrow
  \text{eval(E, D, S, V),}
  \text{ update((D, T), X, V, (D1, T1)),}
  \text{ at(L1, C).}
  \]
Small-Step Semantics

- Encoding correctness when using the Small-Step semantics

\[
\text{false} \leftarrow \text{initConf}(C), \text{reach}(C).
\]
\[
\text{reach}(C) \leftarrow \text{tr}(C, C_1), \text{reach}(C_1).
\]
\[
\text{reach}(C) \leftarrow \text{finalConf}(C).
\]

- VCs generated by using the Small-Step semantics

\[
\text{false} \leftarrow X \geq 1, Y \geq 1, \text{new3}(X, Y).
\]
\[
\text{new3}(X, Y) \leftarrow X = -1, Y = X.
\]
\[
\text{new3}(X, Y) \leftarrow X + 1 = Y, \text{new4}(X, Y).
\]
\[
\text{new3}(X, Y) \leftarrow X \geq 1 + Y, \text{new4}(X, Y).
\]
\[
\text{new4}(X, Y) \leftarrow X \geq Y + 1, \text{new6}(X, Y).
\]
\[
\text{new4}(X, Y) \leftarrow X = Y, \text{new7}(X, Y).
\]
\[
\text{new6}(X, Y) \leftarrow A = X, B = Y, \text{new11}(X, Y, A, B, R).
\]
\[
\text{new7}(X, Y) \leftarrow A = Y, B = X, \text{new8}(X, Y, A, B, R).
\]
\[
\text{new8}(X, Y, A, B, R) \leftarrow R_1 = A - B, \text{new9}(X, Y, A, B, R, R_1).
\]
\[
\text{new9}(X, Y, A, B, R) \leftarrow Y_1 = R, \text{new3}(X, Y_1).
\]
\[
\text{new11}(X, Y, A, B, R) \leftarrow R_1 = A - B, \text{new12}(X, Y, A, B, R, R_1).
\]
\[
\text{new12}(X, Y, A, B, R) \leftarrow X_1 = R, \text{new3}(X_1, Y).
\]

- Linear recursive (at most one atom in the body)

- More predicates and clauses than in Multi-Step semantics VCs
  Multiple predicates for the calls to the \texttt{sub} function (e.g. \text{new11} and \text{new8})

- Half the variables w.r.t. MS semantics VCs
Termination: No sharing cycles

- Algorithm $E$ terminates if
  - the query has no sharing cycles
  - the other clauses have a disjoint, quasi-descending slice decomposition

No multiple occurrences of the same variable in each atom (wlog)

labeled (multi)graph: the nodes are the atoms of the query and there is an edge between two atoms, labeled by variable $X$, iff they share $X$

sharing cycle: path from an atom to itself labeled by distinct variables

\[
\begin{align*}
\text{min}((X, Y, Z) &\rightarrow X > Y, Z > Y) \\
\text{left_drop}(N, \text{leaf}, \text{leaf}) &\leftarrow \\
\text{left_drop}(N, \text{node}(X, L, R), \text{node}(X, L, R)) &\leftarrow N < 1, N1 = N - 1, \text{left_drop}(N1, L, T) \\
\text{false} &\leftarrow N \geq 0, M + N < K, \text{left_drop}(N, \text{leaf}, \text{leaf}), \text{min_leaf}(M), \text{min_leaf}(K)
\end{align*}
\]
Termination: Quasi-descending

- Algorithm $E$ terminates if
  - the query has no sharing cycles
  - the other clauses have a disjoint, quasi-descending slice decomposition

\[
\begin{align*}
\text{min}(X, Y, Z) &\leftarrow X < Y, \ Z = X \\
\text{min}(X, Y, Z) &\leftarrow X \geq Y, \ Z = Y \\
\text{min}_\text{leaf}(\text{leaf}, M) &\leftarrow M = 0 \\
\text{min}_\text{leaf}(\text{node}(X, L, R), M) &\leftarrow M = M_3 + 1, \ \text{min}_\text{leaf}(L, M_1), \ \text{min}_\text{leaf}(R, M_2), \\
\text{min}(M_1, M_2, M_3) \\
\text{left}_\text{drop}(N, \text{leaf}, \text{leaf}) &\leftarrow \\
\text{left}_\text{drop}(N, \text{node}(X, L, R), \text{node}(X, L, R)) &\leftarrow N \leq 0 \\
\text{left}_\text{drop}(N, \text{node}(X, L, R), \text{T}) &\leftarrow N \geq 1, \ N_1 = N - 1, \ \text{left}_\text{drop}(N_1, L, T) \\
\text{false} &\leftarrow N \geq 0, \ M + N < K, \ \text{left}_\text{drop}(N, T, U), \ \text{min}_\text{leaf}(U, M), \ \text{min}_\text{leaf}(T, K)
\end{align*}
\]
Termination: Disjoint slices

- Algorithm $E$ terminates if
  - the query has no sharing cycles
  - the other clauses have a disjoint, quasi-descending slice decomposition

```
min(X, Y, Z) ← X < Y,  Z = X
min(X, Y, Z) ← X ≥ Y,  Z = Y
min_leaf(leaf, M) ← M = 0
min_leaf(node(X, L, R), M) ← M = M3 + 1, min_leaf(L, M1), min_leaf(R, M2), min(M1, M2, M3)
left_drop(N, leaf, leaf) ←
left_drop(N, node(X, L, R), node(X, L, R)) ← N ≤ 0
left_drop(N, node(X, L, R), T) ← N ≥ 1, N1 = N − 1, left_drop(N1, L, T)
false ← N ≥ 0, M + N < K, left_drop(N, T, U), min_leaf(U, M), min_leaf(T, K)
```
A nonterminating transformation

- A property of lists

\[
\text{if } M = N \text{ then } A = Xs
\]

The query has a sharing cycle

\[
\begin{align*}
\text{false } & \leftarrow M = N, \quad \text{take}(M, [\ ], [\ ]), \quad \text{drop}(N, [\ ], [\ ]), \quad \text{append}([\ ], [\ ], A), \quad \text{diff_list}(A, Xs) \\
\text{append}([\ ], Ys, Ys) & \leftarrow \\
\text{append}([X|Xs], Ys, [Z|Zs]) & \leftarrow X = Z, \\
\text{append}(Xs, Ys, Zs) & \leftarrow \\
\text{take}(N, [\ ], [\ ]) & \leftarrow \\
\text{take}(N, [X|Xs], [\ ]) & \leftarrow N = 0 \\
\text{take}(N, [X|Xs], [Y|Ys]) & \leftarrow N \neq 0, \ X = Y, \\
N1 = N - 1, \quad \text{take}(N1, Xs, Ys) & \leftarrow \\
\text{drop}(N, [\ ], [\ ]) & \leftarrow \\
\text{drop}(N, [X|Xs], [Y|Xs]) & \leftarrow N = 0, \ X = Y \\
\text{drop}(N, [X|Xs], Ys) & \leftarrow N \neq 0, \ N1 = N - 1, \\
\text{drop}(N1, Xs, Ys) & \leftarrow \\
\text{diff_list}([\ ], [Y|Ys]) & \leftarrow \\
\text{diff_list}([X|Xs], [\ ]) & \leftarrow \\
\text{diff_list}([X|Xs], [Y|Ys]) & \leftarrow X \neq Y \\
\text{diff_list}(Xs, Ys) & \leftarrow X = Y,
\end{align*}
\]
The Elimination Algorithm \textbf{EC}

- Define new predicates with constraints in LIA or Bool
  - use \textit{widening} operators [Cousot-Halbwachs ‘77, Bagnara et al. ‘08]
- \textbf{EC} guarantees \textit{equisatisfiability}
- If $E$ terminates, then \textbf{EC} terminates
(1) Generate a disjunction $a(U,C)$ of constraints

(2) Check whether or not $LIA \models \forall U. \text{adm}(U) \rightarrow \exists C. a(U,C)$

---

Assume a sound and complete $LIA$-constraint solver: $\text{SOLVE}$. For any set $I_{sp}$ of clauses and query $Q$: $c, A_1, \ldots, A_n$ where $c$ is a $LIA$ constraint,

$\text{SOLVE}(I_{sp}, Q)$ returns

- a satisfiable constraint $a$ s.t. $I_{sp} \cup LIA \models \forall (a \rightarrow Q)$, if any,
- $false$, otherwise

In particular, if $\text{SOLVE}(I_{sp}, \text{reachProp}(U,C)) = a(U,C)$, then

$I_{sp} \cup LIA \models \forall U, C. (a(U,C) \rightarrow \text{reachProp}(U,C))$
\( I_{SP} : \quad q(X) \leftarrow r(X) \)
\[ r(X) \leftarrow X > 0 \]

\text{SOLVE}(I_{SP}, q(X)) \text{ returns the constraint } X > 0

Indeed, \( I_{SP} \cup LIA \models \forall X \ (X > 0 \rightarrow q(X)) \)
(4) Weak Controllability Algorithm

\[
a(U,C) := false; \\
do \{ \\
\quad Q := (\text{reachProp}(U,C) \land \forall C. \neg a(U,C)); 
\quad \text{if (SOLVE}(I_{SP}, Q) = false) \text{ return } false; 
\quad a(U,C) := a(U,C) \lor \text{SOLVE}(I_{SP},Q); 
\} \text{ while } (LIA \vdash \forall U. \text{adm}(U) \rightarrow \exists C. a(U,C)); 
\text{ return } a(U,C); 
\]