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>=0} x=0; y=0; while (x<n) { x=x+1; y=x+y} {y>=x}

Translation

Constrained Horn Clauses
p(X, Y, N) ← N>=0, X=0, Y=0 %Init
p(X1, Y1, N) ← X<N, X1=X+1, Y1=X1+Y, p(X, Y, N) %Loop
false ← X>=N, Y<X, p(X, Y, N) %Exit

- Solution (i.e., model) of the CHCs: p(X, Y, N) ↦ X>=0, Y>=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	$S_0 \longrightarrow S_1 \longrightarrow \ldots \longrightarrow S_n$	Final clauses

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

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.

R1. Definition. Introduce a new predicate definition

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

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

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.

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 ... A :- d_m,G_m in S_i

derive S = { H :- c,d_1,G_1,G ... 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 newp(X) :- c,B. with d→c in Defs

derive D: H :- d,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 newp(X) :- c,B. with Th ⊨ d→c in Defs

derive D: H :- d,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 ⊨ c ↔ 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

given C: H :- d,B,G in S_i newp(X) :- c,B. with Th ⊨ d→c in Defs

derive D: H :- d,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 ⊨ c ↔ 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

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

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 Th $\models$ d$\rightarrow$c in Defs

derive D: H :- d,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

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

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 **false :- c,A**, where **A** is atomic, transform S into a set of clauses S_{SP} such that

  S ∪ {false :- c,A} satisfiable iff S_{SP} ∪ {false :- c,A} satisfiable.

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

  S ∪ {false :- c,G} satisfiable iff S_T ∪ {false :- c,newp(X)} satisfiable.

Specialization Strategy: An Example

```
false :- X<0, p(X,b).                 % ∀X. p(X,b) → X>=0
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X>=0, tm_halts(X).          % the X-th Turing machine halts on X
```

S_0

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).
```

% $\forall X.\ p(X,b) \rightarrow X>=0$ — 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) — S_1

% to a specific constraint on X and value of C

Specialization Strategy: An Example

```
false :- X<0, p(X,b).                 % ∀X. p(X,b) → X>=0                    S0
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X>=0, tm_halts(X).          % 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) S_1

% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b). S_2

~~q(X) :- X<0, X>=0, tm_halts(X).~~ % 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).
```

% $\forall$X. p(X,b) $\rightarrow$ 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) — S_1

% to a specific constraint on X and value of C

Unfold: q(X) :- X<0, X=Y+1, p(Y,b). — S_2

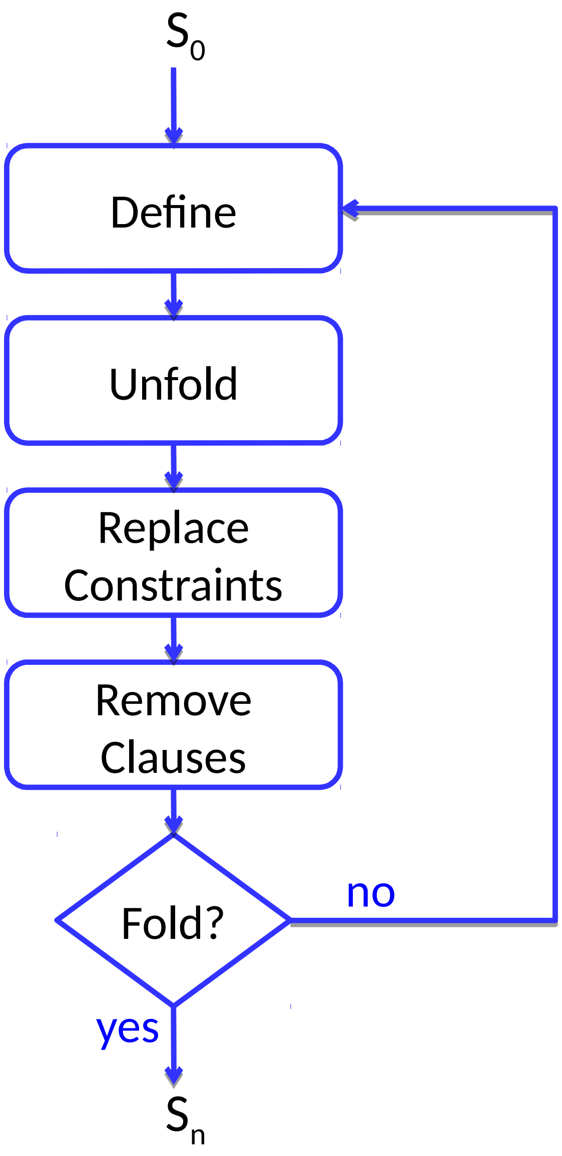
~~q(X) :- X<0, X>=0, tm_halts(X).~~ % clause removal

Fold: false :- X<0, q(X).

q(X) :- X<0, X=Y+1, q(Y). — S_3

Satisfiability of S_3 is easy to check: q(X) ≡ *false* makes all clauses *true* (no facts for q)

A Generic U/F Transformation Strategy

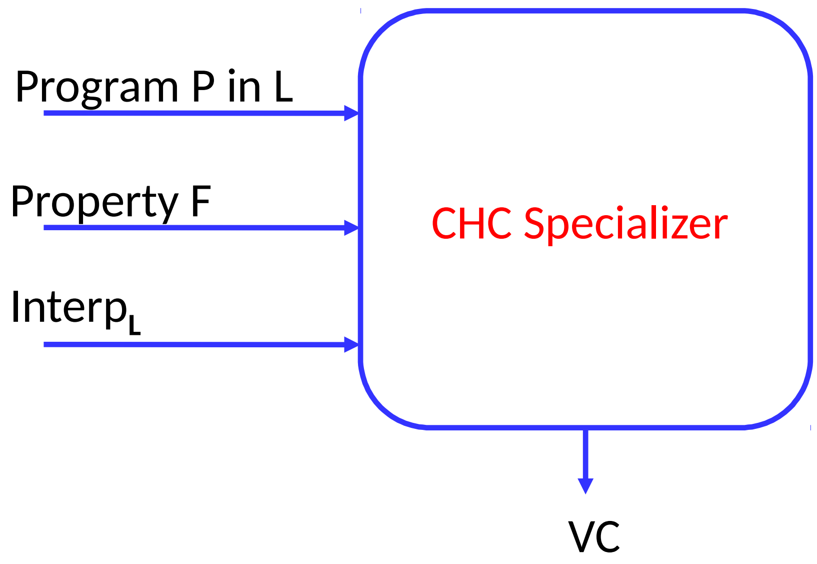


Some Issues About the U/F Strategy

- 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 → g=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



L: Programming language

Interp$_L$: CHC interpreter for L

VC: Verification Conditions, i.e., a set of CHCs independent of L

F holds for P iff VC is satisfiable

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

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: at(Label, Cmd).

```
x=0;
y=0;
while (x<n) {
  x=x+1;
  y=x+y
}
```

0. x=0;
1. y=0;
2. if (x<n) 3 else 6;
3. x=x+1;
4. y=x+y;
5. goto 2;

h. halt

at(**0**,asgn(int(x), int(0))).
at(1,asgn(int(y), int(0))).
at(2, ite(less(int(x), int(n)), 3, 6)).
at(3, asgn(int(x), plus(int(x), int(1)))).
at(4, asgn(int(y), plus(int(x), int(y)))).
at(5, goto(2)).
at(**h**, halt).

A Small-Step Operational Semantics

- The operational semantics is a one-step transition relation between configurations

  <n:cmd, env> ⇒ <n':cmd', env'>

  where: n:cmd is a labelled command
  env is an environment mapping variable identifiers to values

- Assignment

  <n: x=e, env> ⇒ <next(n), update(env, x, [e]env)>

  next(n) is the next labelled command
  update(env, x, [e]env) updates the value of x to the value of expression e in env

- Conditional

  <n: if (e) n1 else n2, env> ⇒ <at(n1), env> if [e]env≠0
  <n: if (e) n1 else n2, env> ⇒ <at(n2), env> if [e]env=0

  at(n) is the labelled command with label n

- Jump

  <n: goto n1, env> ⇒ <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:

  tr(cf(LC1,Env1), cf(LC2,Env2))

CHC Interpreter (Asgn)

assignment x=e;

source configuration target configuration

```
tr( cf(cmd(L, asgn(X,E)), Env1),  cf(cmd(L1, C), Env2)  ) :-
    nextlab(L,L1),             % next label
    at(L1,C),                  % next command
    eval(E,Env1,V),            % evaluate expression
    update(Env1,X,V,Env2).     % update environment
```

More clauses for predicate tr to encode the semantics of the other commands.

Encoding Partial Correctness Properties

- Partial correctness specification (Hoare triple):

  $\{\phi\}$ *prog* $\{\psi\}$

  If the initial values of the program variables satisfy the precondition ϕ and *prog* terminates, **then** the final values of the program variables satisfy the postcondition ψ.
- CHC encoding of partial correctness:

false :- initConf(Cf), errReach(Cf).		
errReach(Cf) :- errorConf(Cf).	PC property	*PC-prop*
errReach(Cf) :- tr(Cf,Cf2), errReach(Cf2).		
initConf(cf(C, Env)) :- at(0,C), ϕ(Env).	Initial configuration	
errorConf(cf(C, Env)) :- at(h,C), $\neg\psi$(Env).	Error configuration	
tr(Cf1,Cf2) :- ...	$Interp_L$	

- $\{\phi\}$ *prog* $\{\psi\}$ is valid iff *PC-prop* is satisfiable.

Problems of direct CHC encoding

- *PC-prop* includes a lot of complex structures and predicates:
  - complex terms encoding configurations:

    cf(cmd(L,asgn(X,Expr)),[(x,1),(y,0),(a,[2,3,4])])
  - recursive predicates over lists encoding functions on the environment:

    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 **cf**, predicates **at**, **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_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>=1} *SumUpto* {y>x}

CHC encoding:

PC-prop

```
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>=1.
errorConf(cf(C, [(x,X),(y,Y),(n,N)])) :- at(h,C), Y≤X.
tr(Cf1,Cf2) :- …
…
at(0,asgn(int(x), int(0))).
…
```

VCGen ↓

Verification Conditions:

VC

```
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.
```

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 $\Rightarrow$ is defined in terms of $\Rightarrow^*$
  - 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 linear wrt the size of the imperative program P
- The size of VC (the number of CHC) is 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

	Small-step (SS^s_f)				Multi-step (MS)				HSF(C)
	QARMC	Z3	MSAT	ELD	QARMC	Z3	MSAT	ELD	
Correct answers	217	208	205	217	210	196	177	182	189
safe problems	161	150	158	158	160	144	147	141	158
unsafe problems	56	58	47	59	50	52	30	41	31
Incorrect answers	5	0	3	2	3	0	1	0	12
false alarms	3	0	1	0	1	0	1	0	3
missed bugs	2	0	2	2	2	0	0	0	9
Timeouts	98	112	112	101	120	124	142	138	119
Total problems	320	320	320	320	320	320	320	320	320
VCG time	221.68	221.68	221.68	221.68	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

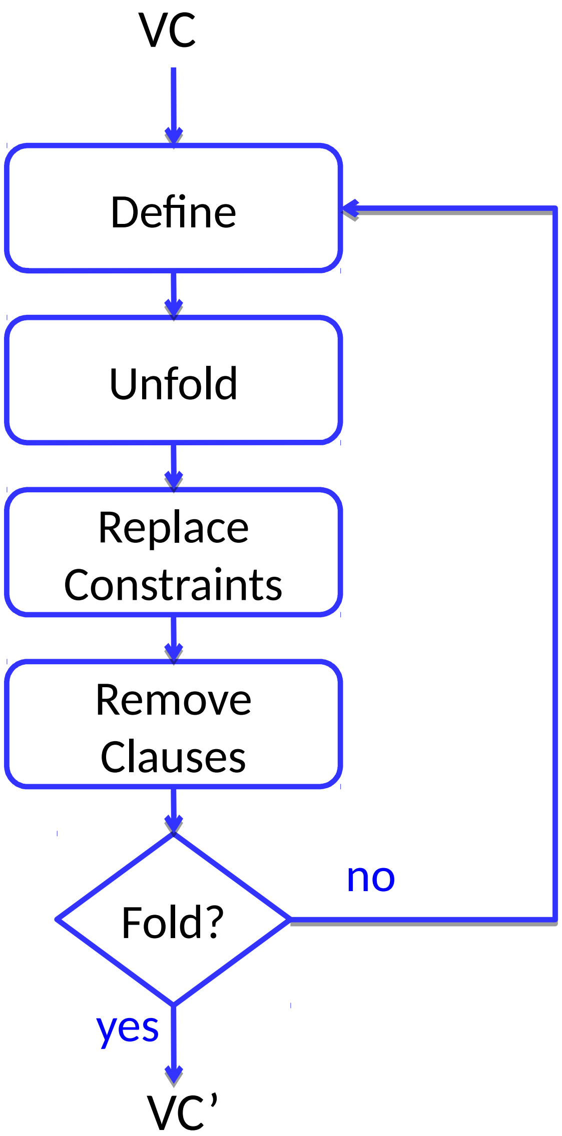
CHC Specialization as CHC Solving

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

VCTransf as CHC Solving

The effect of applying VCTransf can be:

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

2. A set VC' of verification conditions including false :- true.
   VC' is unsatisfiable.

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

propagation of constraint X<0 and constant b

VC

```
false :- X<0, p(X,b).
p(X,C) :- X=Y+1, p(Y,C).
p(X,a).
p(X,b) :- X0, tm_halts(X).
```

VCTransf ⇒

VC'

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

No constrained facts: VC' satisfiable

Iterated CHC Specialization

- If the satisfiability of VC' is unknown *VCTransf* can be *iterated.*
- Between two applications of *VCTransf* we can apply the *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

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

— *Reversal* →

VC'

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

VC is satisfiable iff *VC'* is satisfiable

$$VC_0 \xrightarrow{VCTransf} VC_1 \xrightarrow{Reversal} VC_2 \xrightarrow{VCTransf} VC_3 \;\; \ldots \;\; \xrightarrow{VCTransf} VC_n$$

Iterated CHC Specialization: *SumUpto* Example

VC_0

```
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.
```

Iterated CHC Specialization: *SumUpto* Example

VC_0

```
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.
```

↓ *VCTransf*

VC_1

```
false :- N>=1, 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>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y>=1, N>=1, X>=N, Y<X.
```

Iterated CHC Specialization: *SumUpto* Example

VC_0

```
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.
```

VCTransf

VC_1

```
false :- N>=1, 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>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y>=1, N>=1, X>=N, Y<X.
```

Reversal

VC_2

```
new2(X1, Y1, N) :- N>=1, X1=1, Y1=1.
new3(X1, Y1, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new2(X, Y, N).
new3(X1, Y1, N) :- X1>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, new3(X, Y, N).
false :- N>=1, Y>=1, X>=N, Y<X, new3(X, Y, N).
```

Iterated CHC Specialization: *SumUpto* Example

VC_0

```
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.
```

↓ *VCTransf*

VC_1

```
false :- N>=1, 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>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, new3(X1, Y1, N).
new3(X, Y, N) :- Y>=1, N>=1, X>=N, Y<X.
```

↓ *Reversal*

VC_2

```
new2(X1, Y1, N) :- N>=1, X1=1, Y1=1.
new3(X1, Y1, N) :- X=1, Y=1, N>1, X1=2, Y1=3, new2(X, Y, N).
new3(X1, Y1, N) :- X1>=1, Y1>=X1, X<N, X1=X+1, Y1=X1+Y, new3(X, Y, N).
false :- N>=1, Y>=1, X>=N, Y<X, new3(X, Y, N).
```

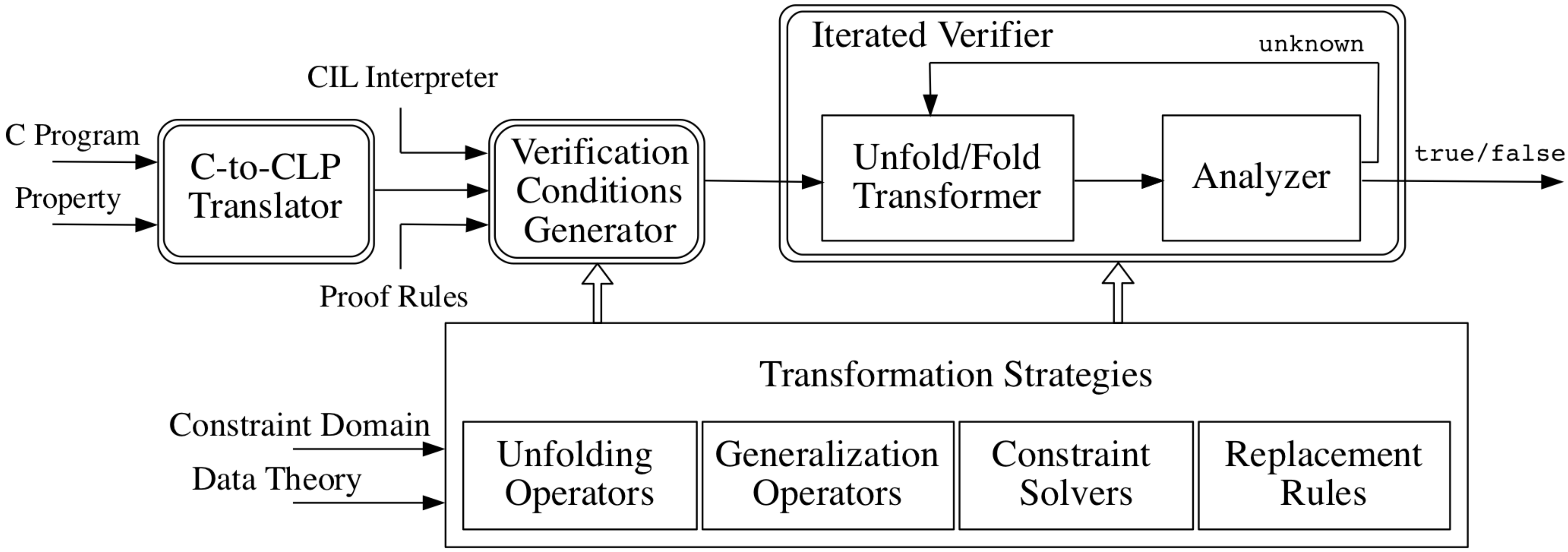
↓ *VCTransf*

VC_3

```
false :- N>=1, Y>=1, X>=N, Y<X, new4(X, Y, N).
```

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) [Grebenshchikov et al. TACAS 2012]
- TRACER [Jaffar, Murali, Navas, Santosa CAV 2012]

		VeriMAP (Gen_{PH})	ARMC	HSF(C)	TRACER	
					SPost	*WPre*
1	*correct answers*	185	138	159	91	103
2	safe problems	154	112	137	74	85
3	unsafe problems	31	26	22	17	18
4	*incorrect answers*	0	9	5	13	14
5	false alarms	0	8	3	13	14
6	missed bugs	0	1	2	0	0
7	*errors*	0	18	0	20	22
8	*timed-out problems*	31	51	52	92	77
9	*total score*	339 (0)	210 (-40)	268 (-28)	113 (-52)	132 (-56)
10	*total time*	10717.34	15788.21	15770.33	27757.46	23259.19
11	*average time*	57.93	114.41	99.18	305.03	225.82

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 read(A,I,V) holds
- if a[i] := v then 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]

read(A,I,X) \ read(A1,J,Y) ⇔ A=A1,I=J | X=Y.

Array-Congruence-2: if a[i]<>a[j] then i<>j

read(A,I,X),read(A1,J,Y) ⇒ A=A1, X<>Y | I<>J.

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

write(A,I,X,A1) \ read(A2,J,Y) ⇔ A1==A2 | (I=J,X=Y) ; (I<>J,read(A,J,Y)).

Array constraint generalization

- Logic variables are decorated with identifiers of the imperative program

: ancestor definition

$\texttt{new3(I,N,A) :- E+1=F, E}\geq\texttt{0, I>F, G}\geq\texttt{H, N>F, N}\leq\texttt{I+1,}$

$\texttt{read(A,E}^{j}\texttt{,G}^{a[j]}\texttt{), read(A,F}^{j1}\texttt{,H}^{a[j1]}\texttt{), reach(I,N,A).}$

: candidate definition

$\texttt{new4(I,N,A) :- E+1=F, E}\geq\texttt{0, I>F, G}\geq\texttt{H, I=1+I1, I1+2}\leq\texttt{C, N}\leq\texttt{I1+3,}$

$\texttt{read(A,E}^{j}\texttt{,G}^{a[j]}\texttt{), read(A,F}^{j1}\texttt{,H}^{a[j1]}\texttt{), read(A,P}^{i}\texttt{,Q}^{a[i]}\texttt{),}$

$\texttt{reach(I,N,A).}$

: **generalized** definition

$\texttt{new5(I,N,A) :- E+1=F, E}\geq\texttt{0, I>F, G}\geq\texttt{H, N>F,}$

$\texttt{read(A,E}^{j}\texttt{,G}^{a[j]}\texttt{), read(A,F}^{j1}\texttt{,H}^{a[j1]}\texttt{), reach(I,N,A).}$

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.

(1) $G = VCGen$								
average time	0.1							
(2) $GZ = VCGen$; Z3								
verification precision	49							
average time	3.5							
(3) $GT = VCGen$; $VCTransf$								
Gen function parameters	$H, I, \Cap$	$H, I, \equiv$	$H, A, \Cap$	$H, A, \equiv$	$W, I, \Cap$	$W, I, \equiv$	$W, A, \Cap$	$W, A, \equiv$
verification precision	60	70	74	71	34	35	34	31
average time	7.8	18.3	5.3	23.6	3.8	10.4	21.1	24.0
(4) $GTZ = VCGen$; $VCTransf$; Z3								
Gen function parameters	$H, I, \Cap$	$H, I, \equiv$	$H, A, \Cap$	$H, A, \equiv$	$W, I, \Cap$	$W, I, \equiv$	$W, A, \Cap$	$W, A, \equiv$
verification precision	67	75	78	75	76	72	80	67
average time	16.8	22.0	8.3	26.3	3.8	7.7	20.2	16.1

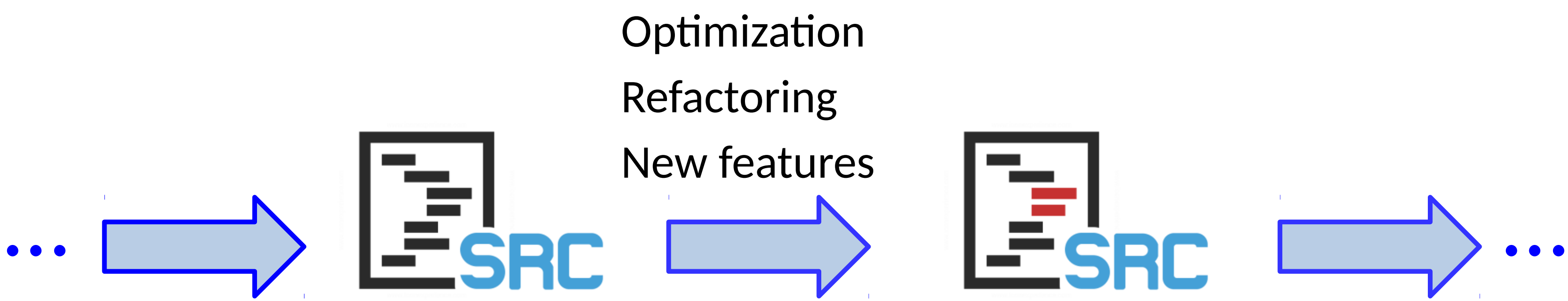
References

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

Verification of relational properties

Relational Properties

- Stepwise program development



- 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

```
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;
}
```

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

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

$$z2 = x2 * y2$$

(Non-tail) recursive

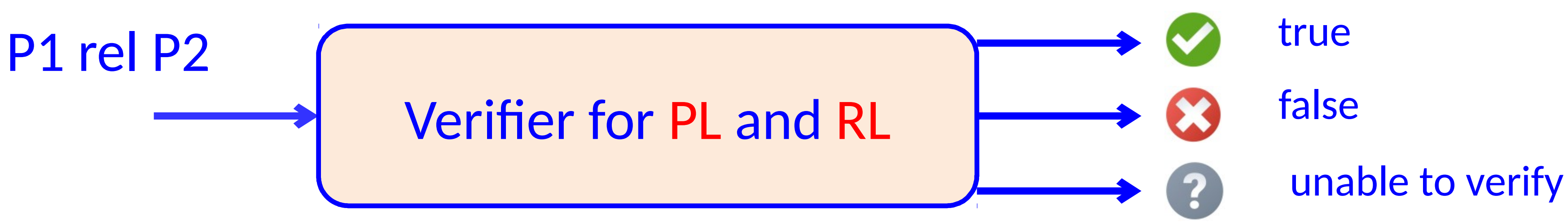
Iterative

- 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

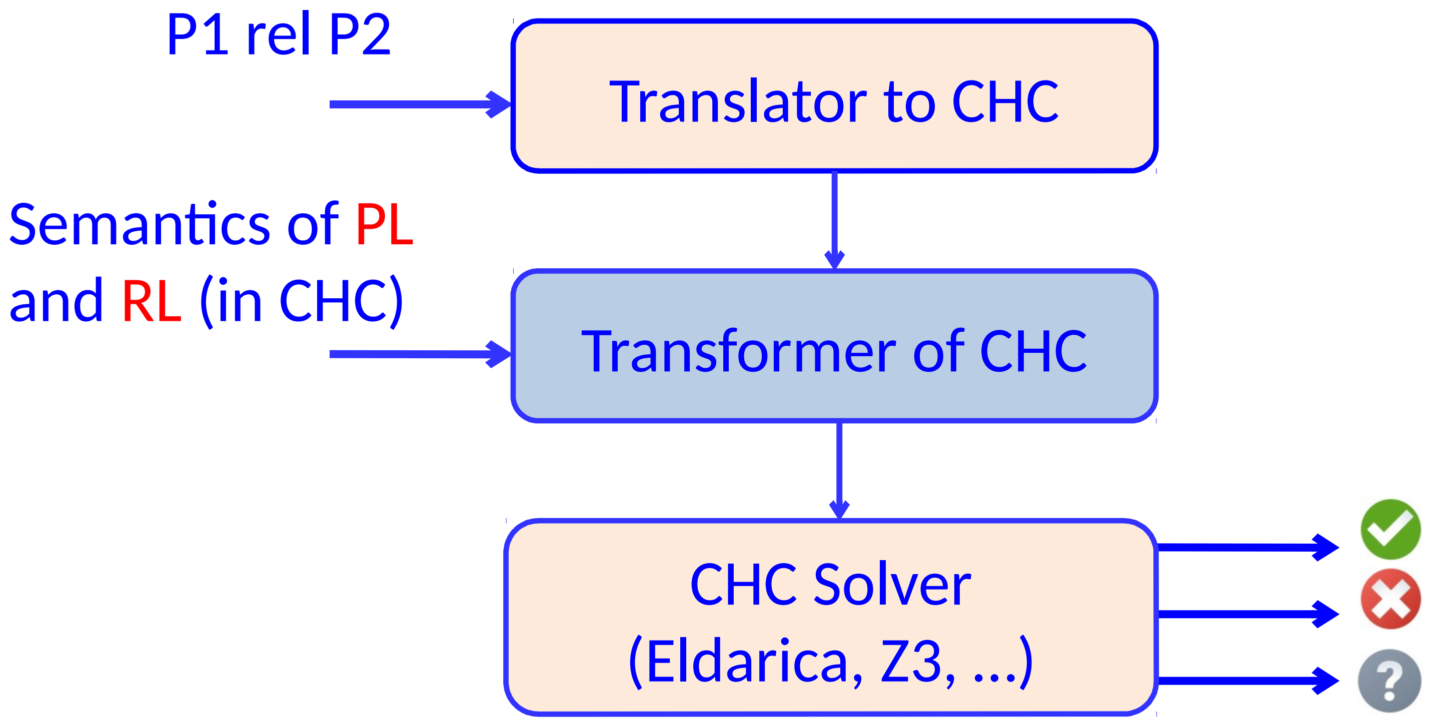
- State-of-the-art verification methods for relational properties are specific for the given programming language PL and class of properties RL [Benton 2004, Barthe *et al.* 2011, Felsing *et al.* 2014]

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



Verification through Horn Clause Transformation

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



Parametric w.r.t. PL and RL.

Relational properties

- Terminating computation

  $\langle P, env_0\rangle \Downarrow env_h$ iff $<l_0{:}c_0, env_0> \Rightarrow^* <l_h{:}halt, env_h>$

- Relational Property P1, P2 programs with disjoint variables, φ,ψ constraints

  $\{\varphi\}\ P1 \sim P2\ \{\psi\}$

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

  if $\vDash \varphi[env_{01} \cup env_{02}]$, $\langle P1, env_{01}\rangle \Downarrow env_{h1}$, $\langle P2, env_{02}\rangle \Downarrow env_{h2}$

  then $\vDash \psi[env_{h1} \cup env_{h2}]$

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;
}
```

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

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

$$z2 = x2 * y2$$

(Non-tail) recursive

Iterative

Relational Property:

{x1=x2 ∧ x2≤y2} `sum_upto` ~ `prod` {z1≤z2}

Encoding the Transition Semantics in CHCs

- Reflexive-transitive closure $\Rightarrow^*$:

  reach(C,C) $\leftarrow$

  reach(C,C2) $\leftarrow$ tr(C,C1), reach(C1,C2)

- Terminating computation $\langle P, env_0 \rangle \Downarrow env_h$ [input/output relation of P]:

  p(X,X') $\leftarrow$ initConf(C,X), reach(C,C'), finalConf(C',X')

  - initConf(C,X): X is the value of the variables in the initial configuration C
  - finalConf(C',X'): X' is the value of the variables in the final configuration C'

Translating Relational Properties into CHCs

- $\{\varphi\}$ P1 ~ P2 $\{\psi\}$

  Prop: false ← pre(X,Y), p1(X,X'), p2(Y,Y'), neg_post(X',Y')

  φ P1 P2 $\neg\psi$

  X,Y,X',Y': *tuples of values for the variables of* P1, P2, resp.

- T_{Prop} = {*Prop*} ∪ {clauses for *p1* and *p2*}

  Correctness of Translation:

  $\{\varphi\}$ P1 ~ P2 $\{\psi\}$ is valid *iff* T_{Prop} is satisfiable

  φ $\neg\psi$

- Example: false ← X1=X2, X2≤Y2, Z1'>Z2',

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

Example Cont'd: CHC Specialization

false ← X1=X2, X2≤Y2, Z1'>Z2',
sum_upto(X1,Z1,X1',Z1'), prod(X2,Y2,Z2,X2',Y2',Z2')

+ clauses for sum_upto and prod

CHC Specializer

Specialized predicates

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)

Limitations of the Specialized CHCs

- To show the satisfiability of

  false ← c(X,Y), p1(X), p2(Y)

  a CHC solver looks for c1(X), c2(Y) such that in T_{SP} ∪ Th:

  p1(X) → c1(X)

  p2(Y) → c2(Y)

  c1(X), c2(Y), c(X,Y) → false

- To show the satisfiability of

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

  a CHC solver has to show that:

  su(X1,Z1') → Z1'≤ 1+ … + X1

  pr(X2,Y2,Z2') → Z2' >= X2∗Y2

  Z1'≤ 1+ … + X1, Z2' >= X2∗Y2, X1=X2, X2≤Y2, Z1'>Z2' → false

- 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)

Predicate Pairing

false ← N≤ Y, W=0, Z1'>Z2',
    fg(N,Z1',Y,W,Z2')
fg(N,Z1',Y,Z2',Z2') ← N≤ 0, Z1'=0
fg(N,Z1', Y,W,Z2') ←
    N>1, N1=N−1, Z1'=R+N, M=Y+W,
    fg(N1,R,Y,M,Z2')

- 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)

false ← c(X,Y), p12(X,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:

  p12(X,Y) → **d**(X,Y)
  d(X,Y), c(X,Y) → false

- **d**(X,Y) captures relations between the variables of p1 and the variables of p2.

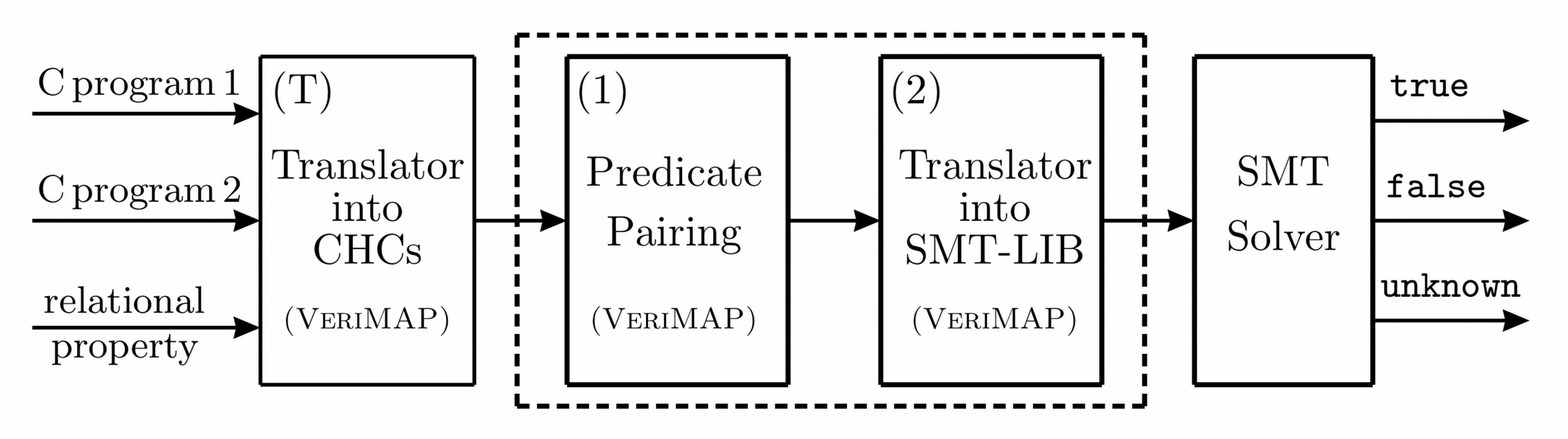
Properties of the CHC transformation rules

- 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 → … → 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



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
  if i1 >= i2 then o1 >= o2
- INJ: injectivity
  if i1 <> i2 then o1 <> o2
- FUN: functional dependency among variables
  if i1 = i2 then o1 = o2
- 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

Problems			Z3 before PP		PP	Z3 after PP	
Category		P	S_1	T_1	T_{PP}	S_2	T_2
(1)	NLIN	13	4	16.11	25.80	13	13.12
(2)	MON	18	1	1.04	2.27	12	3.72
(3)	INJ	11	0	–	1.36	8	1.39
(4)	FUN	7	4	1.39	1.24	7	1.48
(5)	NINT	18	3	0.27	55.80	17	41.33
(6)	LOPT	20	2	4.83	2.98	15	10.71
(7)	ITE	22	5	26.67	4.53	18	17.01
(8)	ARR	6	1	7.45	2.04	5	3.25
(9)	REC	15	6	2.89	1.50	13	4.28
(10)	I-R	4	0	–	0.65	3	1.02
(11)	COMP	10	0	–	16.35	7	6.46
(12)	PCOR	19	5	83.93	17.84	17	17.65
Total number		163	31	144.58	132.36	135	121.42
Average Time				4.66	0.81		0.90

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

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:

```
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: ∀l. listsum(l) >= listmax(l)

Translation into CHCs

- The OCaml program is translated into CHCs:

listsum([],S) ← S=0
listsum([X|Xs],S) ← S=S1+A, abs(X,A), listsum(Xs,S1)
listmax([],M) ← M=0
listmax([X|Xs],M) ← abs(X,A), max(A,M1,M), listmax(Xs,M1)
abs(X,A) ← (X>=0, A=X) **v** (X<0, A= -X)
max(A,M1,M) ← (A>=M1, M=A) **v** (A<M1, M=M1)

- The property is translated into a CHC query:

  false ← S<M, sum(L,S), max(L,M)

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

  listsum(L,S) ↦ (L=[], S=0) **v** (L=[X], abs(X,S)) **v** (L=[X,Y], abs(X,A), abs(Y,B), S=A+B) **v** …
  listmax(L,M) ↦ (L=[], M=0) **v** (L=[X], abs(X,M)) **v** …

Solving CHCs on inductively defined data types by induction

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

  ∀L,S,M. listsum(**L**,S), listmax(**L**,M) → S>=M

  and hence listsum(L,S), listmax(L,M), S<M → false

- Reynolds-Kuncak: Induction for SMT solvers, VMCAI 2015.
- Unno-Torii-Sakamoto: Automating induction for solving Horn clauses, CAV 2017.

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):

```
list-sum-max(S,M) ← S=0, M=0
list-sum-max(S,M) ← S=S1+A, abs(X,A), max(A,M1,M), list-sum-max(S1,M1)
false ← S<M, list-sum-max(S,M)
```

- Solved by Z3, without induction.

  Solution: list-sum-max(S,M) ↦ S>=M, M>=0

- No infinite models are needed to show satisfiabilty

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:

  list-sum-max(S,M) ← listsum(L,S), listmax(L,M)

- Unfold:

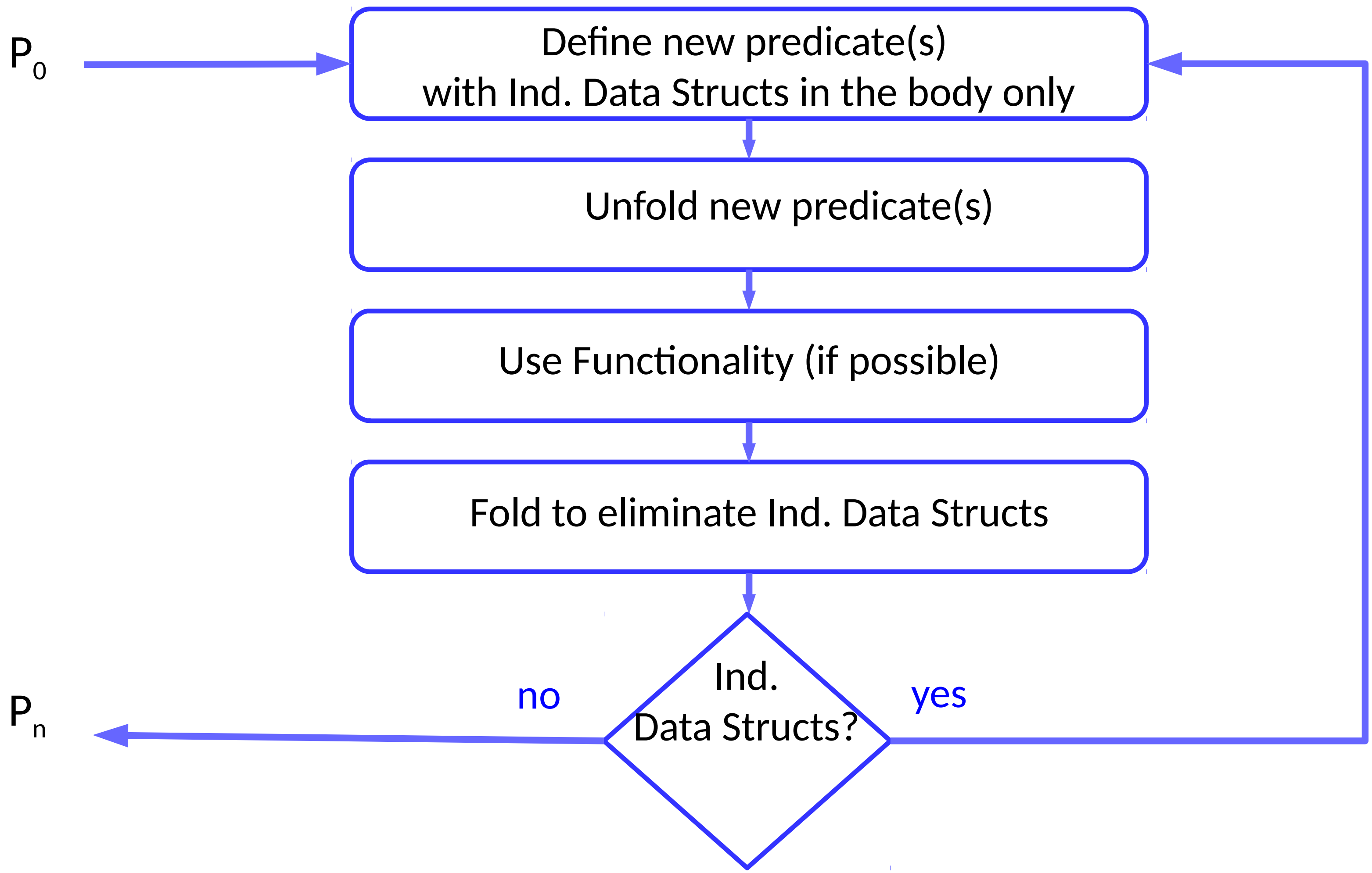
  list-sum-max(S,M) ← S=0, M=0
  list-sum-max(S,M) ← S=S1+A, abs(X,A), max(A,M1,M),
  listsum(Xs,S1), listmax(Xs,M1)

- Fold (eliminate lists):

  list-sum-max(S,M) ← S=0, M=0
  list-sum-max(S,M) ← S=S1+A, abs(X,A), max(A,M1,M),
  list-sum-max(S1,M1)

  false ← S<M, list-sum-max(S,M)

The Elimination Algorithm ***EC***



Termination

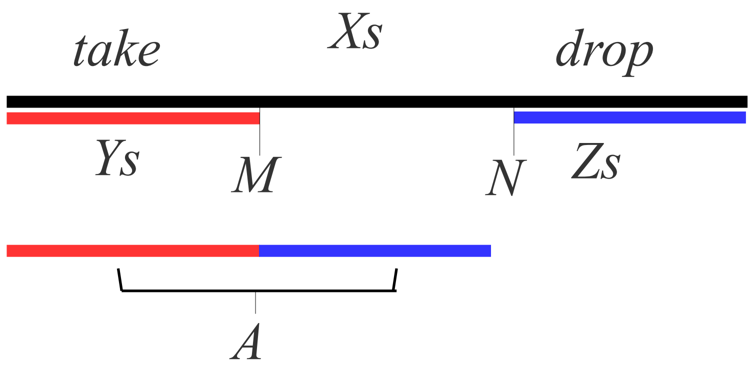
- Algorithm ***E*** terminates if
  - the query has no sharing cycles
  - the other clauses have a disjoint, quasi-descending slice decomposition

$$
\begin{aligned}
& min(X,Y,Z) \leftarrow X < Y,\ Z = X \\
& min(X,Y,Z) \leftarrow X \geq Y,\ Z = Y \\
& min\_leaf(leaf, M) \leftarrow M = 0 \\
& min\_leaf(node(X,L,R), M) \leftarrow M = M3+1,\ min\_leaf(L, M1),\ min\_leaf(R, M2), \\
& \qquad\qquad\qquad\qquad min(M1, M2, M3) \\
& left\_drop(N, leaf, leaf) \leftarrow \\
& left\_drop(N, node(X,L,R), node(X,L,R)) \leftarrow N \leq 0 \\
& left\_drop(N, node(X,L,R), T) \leftarrow N \geq 1,\ N1 = N-1,\ left\_drop(N1, L, T) \\
& false \leftarrow N \geq 0,\ M+N < K,\ left\_drop(N,T,U),\ min\_leaf(U,M),\ min\_leaf(T,K)
\end{aligned}
$$

A nonterminating transformation

- A property of lists

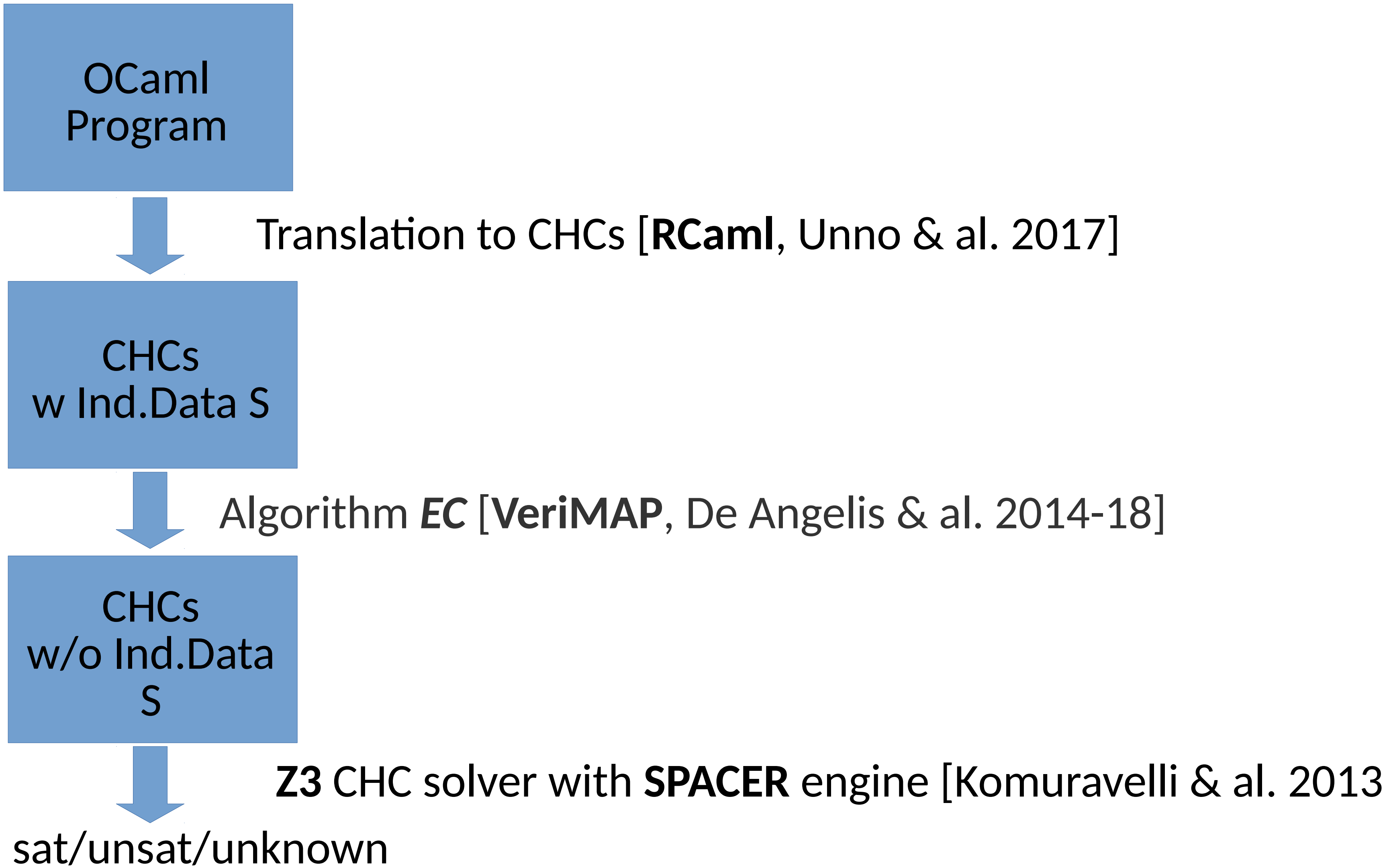
  if $M=N$ then $A=Xs$



$$
\begin{array}{ll}
append([\,], Ys, Ys) \leftarrow & take(N, [\,], [\,]) \leftarrow \\
append([X|Xs], Ys, [Z|Zs]) \leftarrow X=Z, & take(N, [X|Xs], [\,]) \leftarrow N=0 \\
\quad append(Xs, Ys, Zs) & take(N, [X|Xs], [Y|Ys]) \leftarrow N \neq 0,\ X=Y, \\
 & \quad N1 = N-1,\ take(N1, Xs, Ys) \\
drop(N, [\,], [\,]) \leftarrow & diff\_list([\,], [Y|Ys]) \leftarrow \\
drop(N, [X|Xs], [Y|Xs]) \leftarrow N=0, X=Y & diff\_list([X|Xs], [\,]) \leftarrow \\
drop(N, [X|Xs], Ys) \leftarrow N \neq 0, N1 = N-1, & diff\_list([X|Xs], [Y|Ys]) \leftarrow X \neq Y \\
\quad drop(N1, Xs, Ys) & diff\_list([X|Xs], [Y|Ys]) \leftarrow X=Y, \\
 & \quad diff\_list(Xs, Ys)
\end{array}
$$

$$false \leftarrow M=N,\ take(M, Xs, Ys), drop(N, Xs, Zs), append(Ys, Zs, A), diff\_list(A, Xs)$$

Verification of OCaml Programs



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)

Problem Set	n	Z3 S_{Z3}	Z3 T_{Z3}	$\mathcal{EC}$; Z3 $S_{\mathcal{EC};Z3}$	$\mathcal{EC}$; Z3 $T_{\mathcal{EC};Z3}$	RCaml S_{RCaml}	RCaml T_{RCaml}
(1) *FirstOrder*	57	3	0.09	47	37.64	41	216.59
(2) *HigherOrderInstances*	13	1	0.04	11	8.33	10	45.40
(3) *MoreLists*	16	3	13.87	14	11.27	10	119.01
(4) *MoreTrees*	19	5	20.18	19	26.79	5	55.16
Total	105	12	34.18	91	84.03	66	436.17
Avg time			2.85		0.92		6.61

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

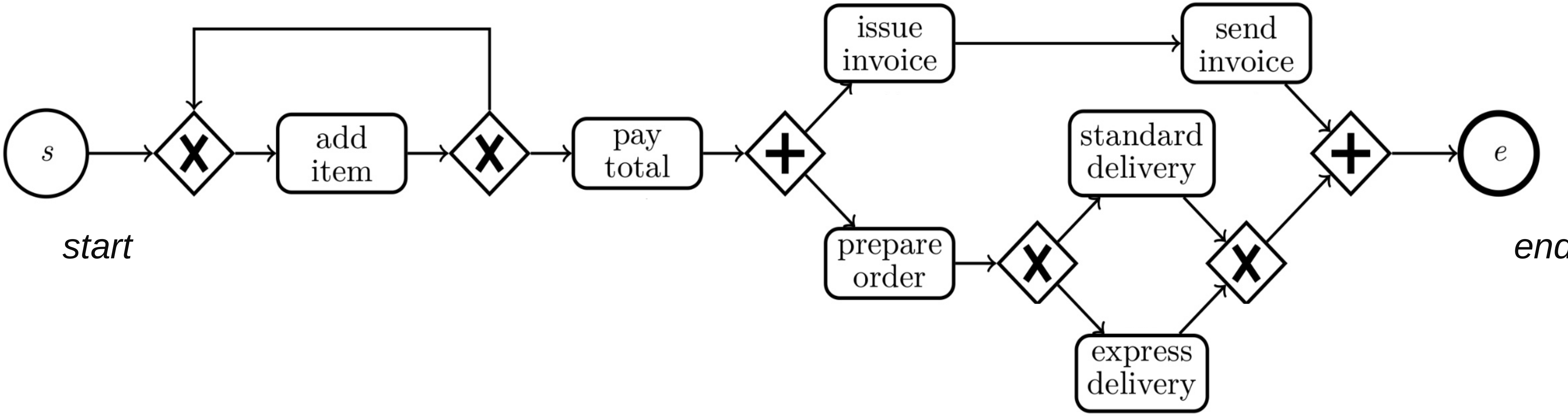
  References

  - [DFPP - TPLP 18]
  - https://fmlab.unich.it/iclp2018/

Verification of time-aware business processes

Business Processes

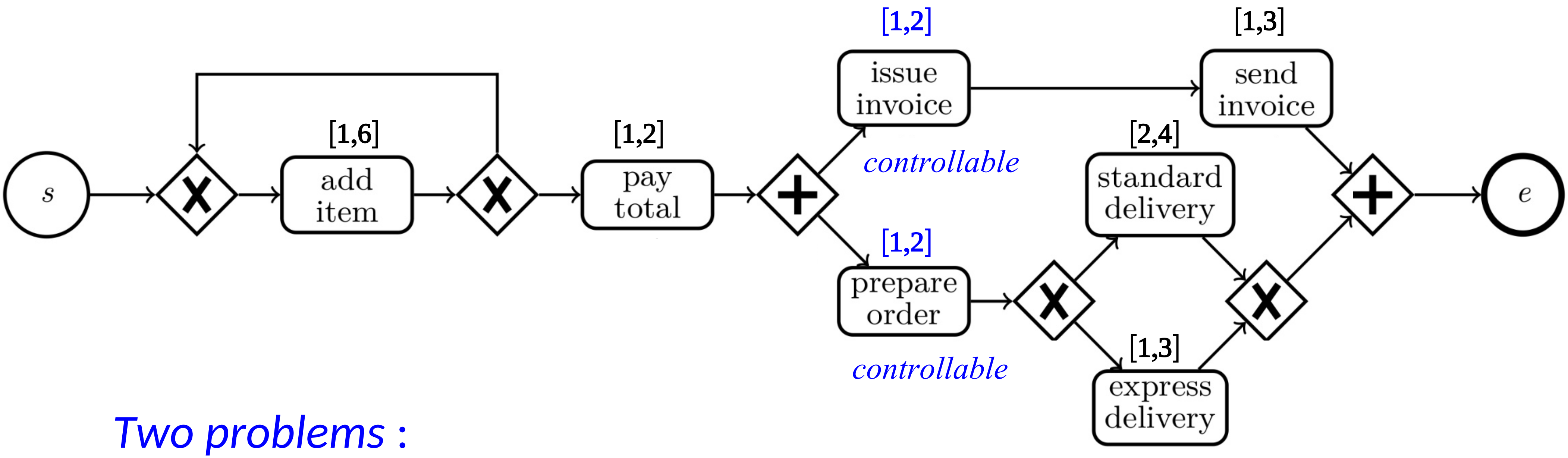
- *Business processes* are 'graphs' for coordinating the activities of an organization towards a business goal.
- *An example: Purchase Order*. A customer adds items to the shopping cart and pays. Then, the vendor issues and sends the invoice, and in parallel, prepares and delivers the order.



There is no information on the durations of tasks.

Time-Aware Business Processes

- *Information on the duration:* Intervals: $d \in [dmin, dmax] \subset \mathbf{N}$



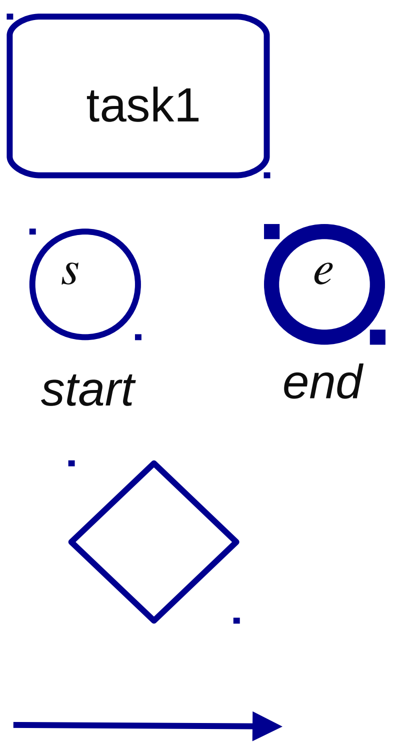
Two problems :

- *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)

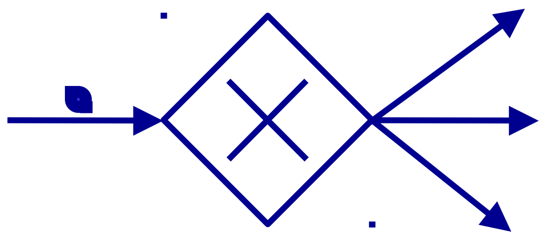
Graphical notation for modeling organizational processes.
BPMN is a standard.

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

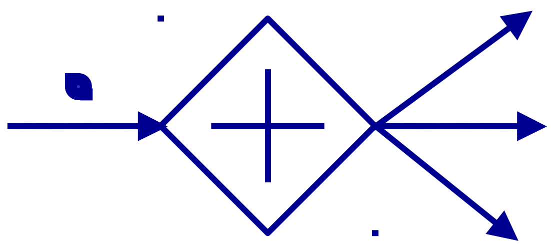


Branch Gateways

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

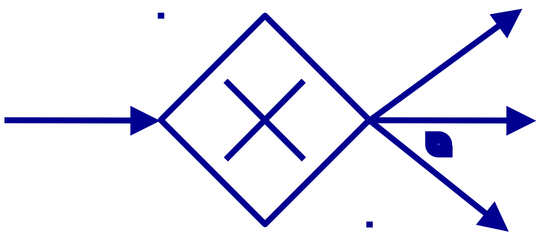


- **parallel** branch gateway (AND)
  - upon activation of the incoming flow *all* outgoing flows are activated

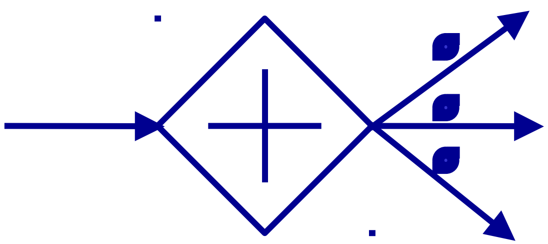


Branch Gateways

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

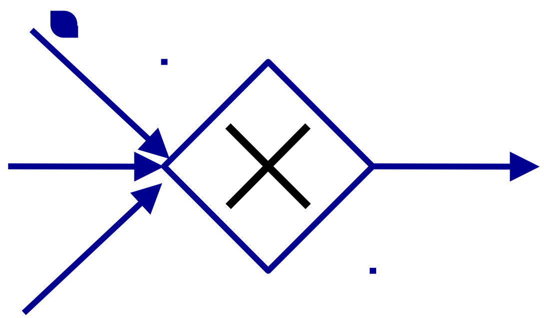


- **parallel** branch gateway (AND)
  - upon activation of the incoming flow *all* outgoing flows are activated

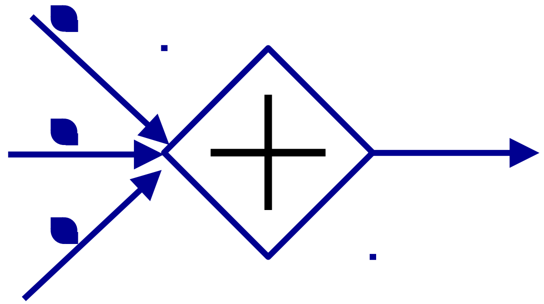


Merge Gateways

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

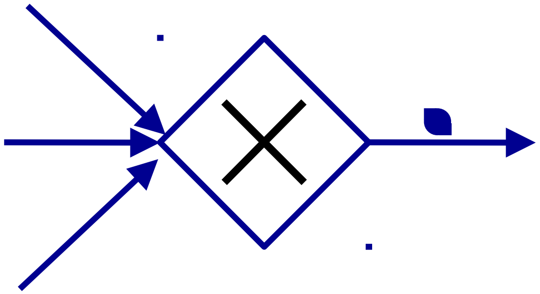


- **parallel** merge gateway (AND)
  - the outgoing flow is activated upon activation of *all* the incoming flows

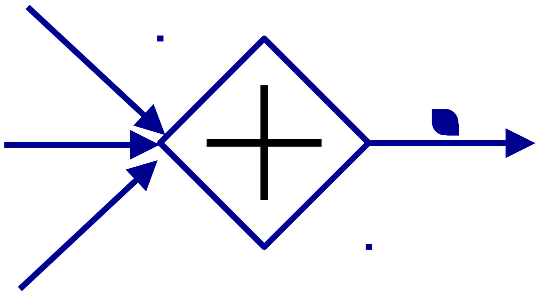


Merge Gateways

- 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

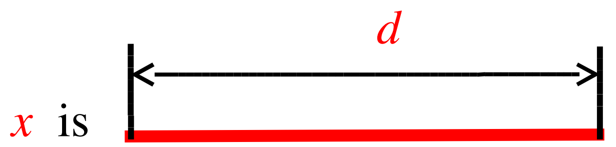
- *seq*(*x*,*y*) — there is an arrow from *x* to *y*
- *t* : time point (i.e., a non-negative integer)

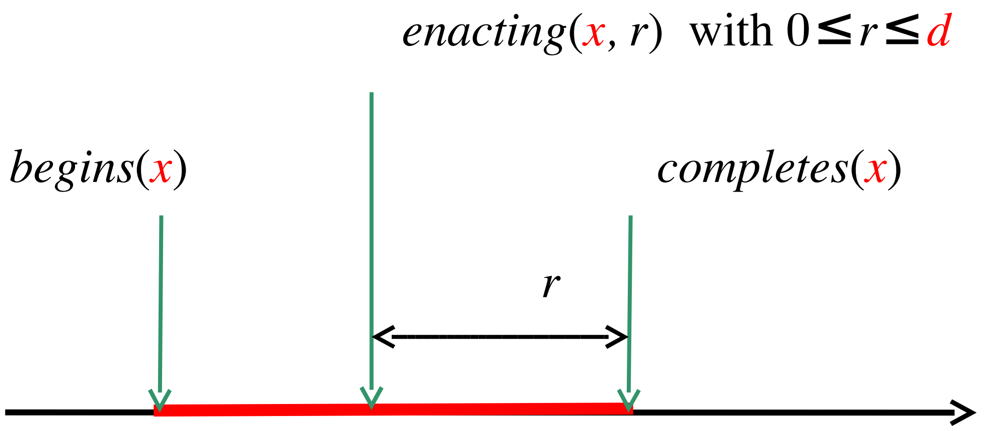
duration(*x*,*d*) — the duration of *x* is *d*

Semantics of time-aware BPMN

$task(x) \leftarrow$

$duration(x, d) \leftarrow 3 \leq d \leq 4$





- durations of events and gateways are assumed to be 0

Semantics of time-aware BPMN

$< F,t > \rightarrow < F',t >$

Instantaneous transition:

$begins(x) \longrightarrow enacting(x, d)$

$$(S_1) \quad \frac{begins(x) \in F \qquad duration(x,d)}{\langle F, t\rangle \longrightarrow \langle (F \setminus \{begins(x)\}) \cup \{enacting(x,d)\},\ t\rangle}$$

Semantics of time-aware BPMN

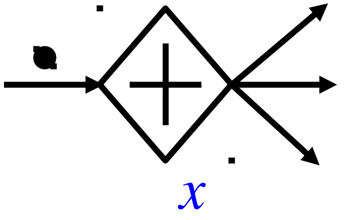
Instantaneous transitions:

< F ,t > → < F',t >

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

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

(S_2) If the parallel branch *x* completes,
then all its successors *s* are enabled, istantaneously



Semantics of time-aware BPMN

Instantaneous transitions:

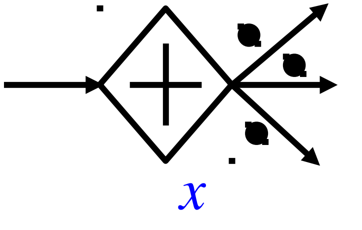
$< F, t > \rightarrow < F', t >$

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

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

(S_2) If the parallel branch x completes,

then all its successors s are enabled, istantaneously



Semantics of time-aware BPMN

$< F,t > \rightarrow < F',t' >$

The time-elapsing transition:

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

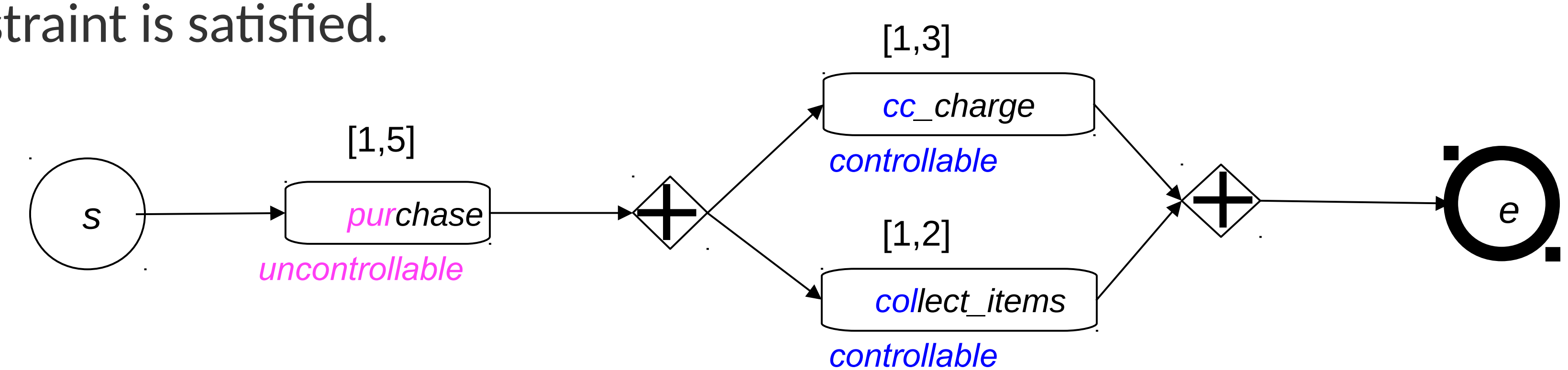
where: (i) $no\_other\_premises(F)$ holds iff none of the premises of rules S_1–S_6 holds, (ii) $m = min\{r \mid enacting(x,r) \in F\}$, (iii) $F \ominus m$ is the set F of fluents where every $enacting(x,r)$ is replaced by $enacting(x,r-m)$, and (iv) $Enbls = \{enables(p,s) \mid enables(p,s) \in F\}$.

Time elapses when no istantaneous 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 *controllable* (e.g., internal to the organization)
  - some tasks are *uncontrollable* (e.g., external to the organization)
- Weak Controllabilty: *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.



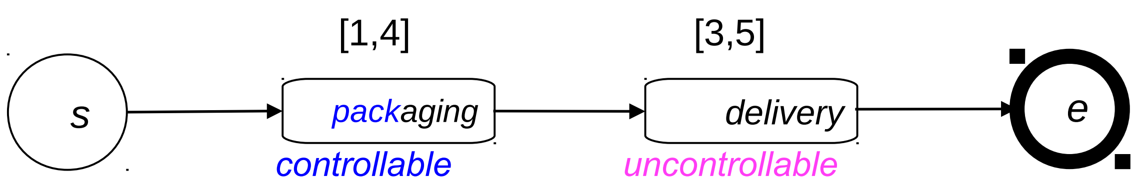
constraint: $3 \leq T_{total} \leq 7$

a solution: *if* $D_{pur}=1$ *then* $D_{cc}=D_{col}=2$ *else* $D_{cc}=D_{col}=1$

Strong Controllability

Weak Controllabilty 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.



constraint: $4 \leq T_{total} \leq 7$

a solution: $1 \leq D_{pack} \leq 2$

CHC translation

Instantaneous transition: $< F,t > \rightarrow < F',t >$

$begins(x) \longrightarrow enacting(x,d)$

$$(S_1) \quad \frac{begins(x) \in F \qquad duration(x,d)}{\langle F,t\rangle \longrightarrow \langle (F \setminus \{begins(x)\}) \cup \{enacting(x,d)\},\ t\rangle}$$

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

where U,C are tuples of uncontrollable and controllable durations, resp.

CHC interpreter of time-aware BPMN

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

$$C2.\ tr(s(F,T),s(FU,T),U,C) \leftarrow select(\{completes(X)\},F),\ par\_branch(X),$$
$$findall(enables(X,S),(seq(X,S)),Enbls),\ update(F,\{completes(X)\},Enbls,FU)$$

$$C3.\ tr(s(F,T),s(FU,T),U,C) \leftarrow select(\{completes(X)\},F),\ not\_par\_branch(X),seq(X,S),$$
$$update(F,\{completes(X)\},\{enables(X,S)\},FU)$$

$$C4.\ tr(s(F,T),s(FU,T),U,C) \leftarrow select(Enbls,F),\ par\_merge(X),$$
$$findall(enables(P,X),(seq(P,X)),Enbls),\ update(F,Enbls,\{begins(X)\},FU)$$

$$C5.\ tr(s(F,T),s(FU,T),U,C) \leftarrow select(\{enables(P,X)\},F),\ not\_par\_merge(X),$$
$$update(F,\{enables(P,X)\},\{begins(X)\},FU)$$

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

$$C7.\ tr(s(F,T),s(FU,TU),U,C) \leftarrow no\_other\_premises(F),\ member(enacting(\_,\_),F),$$
$$findall(Y,(Y=enacting(X,R),\ member(Y,F)),Enacts),$$
$$mintime(Enacts,M),\ M>0,\ decrease\_residual\_times(Enacts,M,EnactsU),$$
$$findall(Z,(Z=enables(P,S),member(Z,F)),Enbls),$$
$$set\_union(Enacts,Enbls,EnactsEnbls),\ update(F,EnactsEnbls,EnactsU,FU),$$
$$TU=T+M$$

CHC translation

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.*

  $RP: \quad reachProp(U,C) \leftarrow c(T,U,C), \quad reach(init, fin(T), U, C)$

  where $c(T,U,C)$ is a constraint

- *Initial state.* $init: \quad < \{begins(start)\}, 0 >$
- *Final state.* $fin(T): \; < \{completes(end)\}, T >$

Encoding Controllability

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*

$$Sem \cup \{RP\} \cup LIA \models \forall U.\ adm(U) \rightarrow \exists C\ reachProp(U,C)$$

where *adm*(*U*) iff the durations in *U* belong to the given intervals

- *Strong Controllability*

$$Sem \cup \{RP\} \cup LIA \models \exists C.\ \forall U.\ adm(U) \rightarrow reachProp(U,C)$$

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 ∃–∀ and ∀–∃
  - 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 ∃–∀ and ∀–∃ with recursive clauses to
  - computing answers to queries
  - solving a set of quantified LIA contraints

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
  [Vidal-Fargier 1999, Combi-Posenato 2009, Cimatti et al. 2015, Zavatteri et al. 2017]

- 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 x=f(e1,...,en); “return” case

```
tr(cf(cmd(L,asgn(X,call(F,Es))), (D,S)),          source configuration
   cf(cmd(L2,C2),                (D2,S2)))         target configuration

←
    eval_list(Es,D,S,Vs),                          evaluate function parameters
    build_funenv(F,Vs,FEnv),                       build function environment
    firstlab(F,FL),  at(FL,C),                     first label and command function def
    reach( cf(cmd(FL,C),       (D,FEnv)),          function execution
           cf(cmd(LR,return(E)),(D1,S1))),         return
    eval(E,(D1,S1),V),                             evaluate returned expression
    update((D1,S),X,V,(D2,S2)),                    update caller environment
    nextlab(L,L2), at(L2,C2)                       next label and command
```

Multi-Step Operational Semantics

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)          loop execution
new3(X,Y, X1,Y1) ← X>=Y+1, new4(X,Y, X1,Y1)          loop execution
new3(X,Y, X,Y) ← X=Y                                 loop exit
new4(X,Y, X3,Y3) ← X>=Y+1, A=X, B=Y, X2=R1,          then branch
                   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,         else branch
                    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                sub  function call
```

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

```
tr(cf(cmd(L,asgn(X,call(F,Es))),D,T),
   cf(cmd(FL,C),                    D,[frame(L1,X,Fenv)|T]))   ←
         nextlab(L,L1),
         loc_env(T,S),  eval_list(Es,D,S,Vs),
         build_funenv(F,Vs,FEnv),
         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

```
tr(cf(cmd(L,return(E)),D, [frame(L1,X,S) |T]),
   cf(cmd(L1,C),        D1,T1))                       ←
         eval(E,D,S,V),
         update((D,T),X,V,(D1,T1)),
         at(L1,C).
```

Small-Step Semantics

- Encoding correctness when using the Small-Step semantics

false ← initConf(C), reach(C).
reach(C) ← tr(C,C1), reach(C1).
reach(C) ← finalConf(C).

- VCs generated by using the Small-Step semantics

```
false ← X>=1, Y>=1, new3(X,Y).
new3(X,Y) ← X=<-1, Y=X.
new3(X,Y) ← X+1=<Y, new4(X,Y).
new3(X,Y) ← X>=1+Y, new4(X,Y).
new4(X,Y) ← X>=Y+1, new6(X,Y).
new4(X,Y) ← X=<Y, new7(X,Y).
```

```
new6(X,Y) ← A=X, B=Y, new11(X,Y,A,B,R).
new7(X,Y) ← A=Y, B=X, new8(X,Y,A,B,R).
new8(X,Y,A,B,R) ← R1=A-B, new9(X,Y,A,B,R1).
new9(X,Y,A,B,R) ← Y1=R, new3(X,Y1).
new11(X,Y,A,B,R) ← R1=A-B, new12(X,Y,A,B,R1).
new12(X,Y,A,B,R) ← X1=R, new3(X1,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 sub function (e.g. new11 and new8)
- Half the variables w.r.t. MS semantics VCs

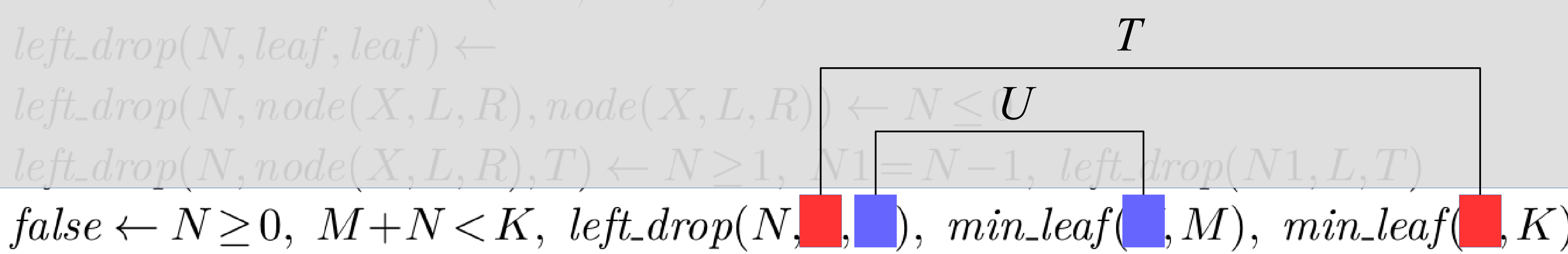
Termination: No sharing cycles

- 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



Termination: Quasi-descending

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

Slice: take one "inductive" argument for each predicate

Quasi-descending: body arguments are (possibly non-strict) subterms of head arguments

$$min(X,Y,Z) \leftarrow X<Y,\ Z=X$$
$$min(X,Y,Z) \leftarrow X\geq Y,\ Z=Y$$
$$min\_leaf(leaf,M) \leftarrow M=0$$
$$min\_leaf(node(X,L,R),M) \leftarrow M=M3+1,\ min\_leaf(L,M1),\ min\_leaf(R,M2),\ min(M1,M2,M3)$$
$$left\_drop(N,leaf,leaf) \leftarrow$$
$$left\_drop(N,node(X,L,R),node(X,L,R)) \leftarrow N\leq 0$$
$$left\_drop(N,node(X,L,R),T) \leftarrow N\geq 1,\ N1=N-1,\ left\_drop(N1,L,T)$$
$$false \leftarrow N\geq 0,\ M+N<K,\ left\_drop(N,T,U),\ min\_leaf(U,M),\ min\_leaf(T,K)$$

Termination: Disjoint slices

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

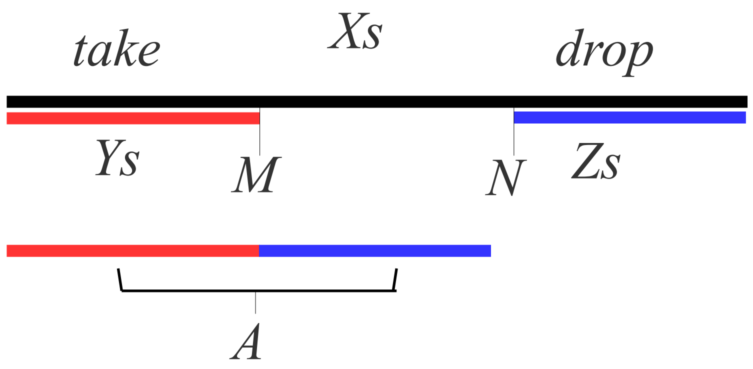
Disjoint: no variable is shared between two slices of the same clause

$$min(X,Y,Z) \leftarrow X<Y,\ Z=X$$
$$min(X,Y,Z) \leftarrow X \geq Y,\ Z=Y$$
$$min\_leaf(leaf, M) \leftarrow M=0$$
$$min\_leaf(node(X,L,R), M) \leftarrow M=M3+1,\ min\_leaf(L,M1),\ min\_leaf(R,M2),\ min(M1,M2,M3)$$
$$left\_drop(N, leaf, leaf) \leftarrow$$
$$left\_drop(N, node(X,L,R), node(X,L,R)) \leftarrow N \leq 0$$
$$left\_drop(N, node(X,L,R), T) \leftarrow N \geq 1,\ N1=N-1,\ left\_drop(N1,L,T)$$
$$false \leftarrow N \geq 0,\ M+N<K,\ left\_drop(N,T,U),\ min\_leaf(U,M),\ min\_leaf(T,K)$$

A nonterminating transformation

- A property of lists

  if $M\!=\!N$ then $A\!=\!Xs$

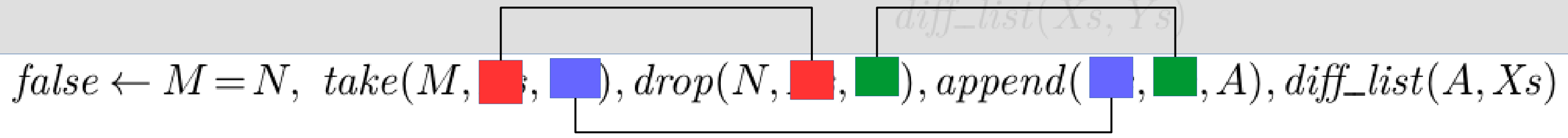


$append([\,], Ys, Ys) \leftarrow$
$append([X|Xs], Ys, [Z|Zs]) \leftarrow X = Z,$
$\quad append(Xs, Ys, Zs)$

$drop(N, [\,], [\,]) \leftarrow$
$drop(N, [X|Xs], [Y|Xs]) \leftarrow N = 0, X = Y$
$drop(N, [X|Xs], Ys) \leftarrow N \neq 0, N1 = N-1,$
$\quad drop(N1, Xs, Ys)$

$take(N, [\,], [\,]) \leftarrow$
$take(N, [X|Xs], [\,]) \leftarrow N = 0$
$take(N, [X|Xs], [Y|Ys]) \leftarrow N \neq 0, \; X = Y,$
$\quad N1 = N-1, \; take(N1, Xs, Ys)$
$diff\_list([\,], [Y|Ys]) \leftarrow$
$diff\_list([X|Xs], [\,]) \leftarrow$
$diff\_list([X|Xs], [Y|Ys]) \leftarrow X \neq Y$
$diff\_list([X|Xs], [Y|Ys]) \leftarrow X = Y,$
$\quad diff\_list(Xs, Ys)$

The query has a sharing cycle



The Elimination Algorithm ***EC***

- Define new predicates with constraints in LIA or Bool
  - use widening operators [Cousot-Halbwachs '77, Bagnara et al. '08]
- ***EC*** guarantees equisatisfiability
- If ***E*** terminates, then ***EC*** terminates

(4) Weak Controllability Algorithm

(1) Generate a disjunction $a(U,C)$ of constraints

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

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

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 SOLVE(I_{SP}, $reachProp(U,C)$) = $a(U,C)$, then

$$I_{SP} \cup LIA \models \forall U,C.\ (a(U,C) \rightarrow reachProp(U,C))$$

(4) Weak Controllability Algorithm

I_{SP} : $q(X) \leftarrow r(X)$

$r(X) \leftarrow X>0$

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

Indeed, $I_{SP} \cup LIA \models \forall X \; (X>0 \rightarrow q(X))$

(4) Weak Controllability Algorithm

$a(U,C) := \mathit{false};$
do {

$Q := (\text{reachProp}(U,C) \land \forall C.\ \neg a(U,C));$

if $(\text{SOLVE}(I_{SP}, Q) = \mathit{false})$ *return* *false*;

$a(U,C) := a(U,C) \lor \text{SOLVE}(I_{SP}, Q);$

} *while* $(LIA \not\models \forall U.\ adm(U) \rightarrow \exists C.\ a(U,C))$;
return $a(U,C);$