

# Modeling Systems in CLP

Joxan Jaffar

Andrew E. Santosa

Răzvan Voicu

School of Computing, National University of Singapore  
S16, 3 Science Drive 2, Singapore 117543

## Abstract

We present a methodology for the modeling of complex program behavior in CLP. In the first part we present an informal description about how to represent a system in CLP. At its basic level, this representation captures the trace semantics of concurrent programs, or even high-level specifications, in the form of a predicate transformer. Based on traces, the method can also capture properties of the underlying runtime system such as the scheduler and the microarchitecture, so as to provide a foundation for reasoning about resources such as time and space.

The second part presents a formal and compositional proof method for reasoning about safety properties of the underlying system. The idea is that a safety property is simply a CLP goal, and its proof is established by executing the goal by a CLP interpreter. However, a traditional CLP interpreter does not suffice. We thus introduce a technique of *coinductive tabling* to CLP. Essentially, this extends CLP so that it can inductively use proof obligations that are assumed but not yet proven, and it can generate new proof obligations assertions dynamically.

## 1 Outline

We present a general CLP approach for program verification:

- We model a wide range of programs and systems in CLP
- We present a proof method to reason about any CLP program

Poster structure:

- Modeling (Sec. 2)
  - Symbolic representation of transition relations in CLP
  - Modeling examples (Sec. 3)
    - \* Sequential program
    - \* Concurrent program: synchronous and asynchronous composition
    - \* Scheduling
    - \* Resource constraint: timing
    - \* Microarchitecture: instruction cache
    - \* High-level specifications: a non-standard timed automata, State-chart
- Proof method based on coinductive tabling for proving safety of general CLP programs (Sec. 4)
- Ongoing works (Sec. 5)
  - Intermittent abstraction
  - Liveness

## 2 Modeling in CLP

- We model a program  $P$  with variables  $\tilde{X}$  as a *predicate transformer* by first identifying *target variables*  $\tilde{X}^t$  corresponding to  $\tilde{X}$ , and then establishing a constraint on  $\tilde{X}$  and  $\tilde{X}^t$ .
- To outline the predicate transformer aspect of a (possibly nondeterministic) program, we express its semantics as a set of CLP clauses that define the predicate  $\text{state}(PP, \tilde{X}, \tilde{X}^t)$ , where the logic variables  $\tilde{X}$  and  $\tilde{X}^t$  represent values of program variables at program point  $PP$ , and at a target program point, respectively.
- The  $\text{state}$  predicate realizes the following relation: if  $\tilde{X}$  are values of variables at program point  $PP$ , then  $\tilde{X}^t$  are possible values of the program variables at the target program point.

### 2.1 Bottom-Up Vs. Top-Down

- Consider the simple program  $\langle 0 \rangle x := x + 1 \langle 1 \rangle$  where 0 denotes the entry point and 1 denotes the exit point. The *bottom-up* CLP model of this program is:

```
state(0, X, Xt) :- state(1, X + 1, Xt).
state(1, X, X).
```

- Running a goal such as  $?- \text{state}(0, X, Xt), X > 5$ , would return  $Xt > 6$ , the strongest postcondition of  $P$  when run with input  $x > 5$ .
- Similarly, the *top-down* CLP model:

```
state(1, X, Xt) :- state(0, X - 1, Xt).
state(0, X, X).
```

captures the weakest precondition of  $P$ .

- Running  $?- \text{state}(1, X, Xt), X > 5$ , would return  $Xt > 4$ .
- Can also represent properties of *nonterminating* programs. Consider for example the program  $\langle 0 \rangle \text{while}(\text{true}) x := x + 1 \langle 1 \rangle \text{endwhile} \langle 2 \rangle$  where the program points 0, 1 and 2 are displayed. The bottom-up CLP model:

```
state(0, X, Xt) :- state(1, X + 1, Xt).
state(1, X, Xt) :- state(1, X + 1, Xt).
state(1, X, X).
```

captures in  $Xt$  all the values of  $X$  that can appear at program point 1. That is, the goal  $?- \text{state}(0, X, Xt), X > 5$ , for example, has as solutions:  $Xt > N$  for all  $N > 6$ .

## 3 Example Modeling

### 3.1 Sequential Program

```
 $\langle 0 \rangle$  while (i < n) do
 $\langle 1 \rangle$    i := i + 1
 $\langle 2 \rangle$  end  $\langle 3 \rangle$ 
```

CLP representation (bottom-up):

```
state(0, I, N, It, Nt) :- I=0, N>=0.
state(1, I, N, It, Nt) :- I<N, state(0, I, N, It, Nt).
state(1, I, N, It, Nt) :- I<N, state(2, I, N, It, Nt).
state(2, I+1, N, It, Nt) :- state(1, I, N, It, Nt).
state(3, I, N, It, Nt) :- I>=N, state(0, I, N, It, Nt).
state(3, I, N, It, Nt) :- I>=N, state(2, I, N, It, Nt).
```

CLP representation (top-down):

```
state(0, I, N, It, Nt) :- I>=N, state(3, I, N, It, Nt).
state(0, I, N, It, Nt) :- I<N, state(1, I, N, It, Nt).
state(1, I, N, It, Nt) :- state(2, I+1, N, It, Nt).
state(2, I, N, It, Nt) :- I<N, state(3, I, N, It, Nt).
state(2, I, N, It, Nt) :- I>=N, state(3, I, N, It, Nt).
state(3, I, N, I, N).
```

- A while program with the target program points are 3, for the bottom-up model, and 0 for the top-down model.

### 3.2 Concurrent Program

```
while (true) do
 $\langle 0 \rangle$    t1 := t2 + 1
 $\langle 1 \rangle$    await (t1 < t2  $\vee$  t2=0)
 $\langle 2 \rangle$    t1 := 0
end
while (true) do
 $\langle 0 \rangle$    t2 := t1 + 1
 $\langle 1 \rangle$    await (t2 < t1  $\vee$  t1=0)
 $\langle 2 \rangle$    t2 := 0
end
```

- A two-process bakery mutual exclusion algorithm with the point  $\langle 2 \rangle$  indicating the critical section. Initially,  $t1 = t2 = 0$ .
- This is an infinite-state program suitable for CLP approach.
- We consider “blocking” concurrency here where we have an “await” statement, which blocks until the specified condition holds.
- In this example, we adopted an asynchronous composition (instructions interleaving) of processes, but synchronous composition can also be modeled.

### CLP representation (top-down):

```
state([0,0], T1,T2) :- T1=0, T2=0.
state([1,P2], T1',T2) :- T1'=T2+1,
    state([0,P2], T1,T2).
state([2,P2], T1,T2) :- (T1<T2; T2=0),
    state([1,P2], T1,T2).
state([0,P2], T1',T2) :- T1'=0, state([2,P2], T1,T2).
state([P1,1], T1,T2') :- T2'=T1+1,
    state([P1,0], T1,T2).
state([P1,2], T1,T2) :- (T2<T1; T1=0),
    state([P1,1], T1,T2).
state([P1,0], T1,T2') :- T2'=0, state([P1,2], T1,T2).
```

- As in this example, we often remove the target variables.
- The safety property of interest, mutual exclusion, can be obtained by proving that  $?- \text{state}([2,2], T1, T2)$  has no solutions.

### 3.3 Scheduling

```
while (true) do
<0>   x := x + 1
end
while (true) do
<0>   y := y + 1
end
```

- Here we model not just the visible aspect of the user program, but also the underlying scheduling mechanism.
- Process 1 executes at least 1 and at most 3 instructions before control is passed to Process 2. Thus we implement “ $k$ -fairness” where  $k = 3$ .

### CLP representation (top-down):

```
state([0,0],Q,X,Y) :- Q=0,X=0,Y=0.
state([0,P2],Q+1,X+1,Y) :- Q<=2, state([0,P2],Q,X,Y).
state([P1,0],0,X,Y+1) :- Q>0, state([P1,0],Q,X,Y).
```

- Variable  $Q$  represents the state of the scheduler.
- $Q$  is incremented whenever Process 1 executes, but Process 1 can only execute while  $Q \leq 2$ .
- On the other hand, execution of Process 2 is only possible when  $Q > 0$ , that is when Process 1 has been executed after last execution of Process 2, and this resets  $Q$  to 0.
- We can prove, for example, that  $\text{state}([P1,P2],Q,X,Y)$  implies  $X \leq Y \times 3$ .

### 3.4 Timing

```
<0> x := 2
<1> while (x <= n) do
<2>   a[x]:=a[x-1]+a[x-2]
<3>   x:=x+2
end <4>
<0> y := 3
<1> delay(300)
<2> while (y <= n) do
<3>   a[y]:=a[y-1]+a[y-2]
<4>   y:=y+2
end <5>
```

- A parallel Fibonacci program where processes run on separate processors, and access a shared array  $a$ . Initially  $a[0] = 0$ ,  $a[1] = 1$ , and  $a[i] = 0$  for all  $i \geq 2$ .
- Dangerous because array element can be accessed simultaneously, but with the right timings, the program remains correct to an extent.
- We assume that every transition takes a fixed number  $\epsilon$  of time units, where  $95 \leq \epsilon \leq 105$ . Under this assumption, when  $N \leq 3$ , both process never access the same array location.
- The notation  $A[I]$  to denote the  $I$ -th element of array  $A$ , and  $\langle A, I, J \rangle$  to denote the array resulting from replacing the  $I$ -th element in  $A$  by  $J$ .
- We can verify the goal  $?- \text{state}([4,5],T1,T2,A,X,Y,N), N \leq 3$  implies  $A[N] = \text{fib}(N)$ .

### CLP representation (top-down):

```
state([0,0],T1,T2,A,X,Y,N) :- T1=0, T2=0.
state([1,P2],T1',T2,A,2,Y,N) :-
    inc(T1,T2,T1'), state([0,P2],T1,T2,A,X,Y,N).
state([2,P2],T1',T2,A,X,Y,N) :-
    inc(T1,T2,T1'), X<=N, state([1,P2],T1,T2,A,X,Y,N).
state([4,P2],T1',T2,A,X,Y,N) :-
    inc(T1,T2,T1'), X > N, state([1,P2],T1,T2,A,X,Y,N).
state([3,P2],T1',T2,A',X,Y,N) :- inc(T1,T2,T1'),
    A'=<A,X,A[X-1]+A[X-2]>, state([2,P2],T1,T2,A,X,Y,N).
state([1,P2],T1',T2,A,X+2,Y,N):-
    inc(T1,T2,T1'),state([3,P2],T1,T2,A,X,Y,N).
state([P1,1],T1,T2',A,X,3,N) :- inc(T2,T1,T2'),
    state([P1,0],T1,T2,A,X,Y,N).
state([P1,2],T1,T2',A,X,Y,N) :-
    T2<=T1,T2'=T2+300, state([P1,1],T1,T2,A,X,Y,N).
state([P1,3],T1,T2',A,X,Y,N) :-
    inc(T2,T1,T2'), Y<=N, state([P1,2],T1,T2,A,X,Y,N).
state([P1,5],T1,T2',A,X,Y,N) :-
    inc(T2,T1,T2'), Y>N, state([P1,2],T1,T2,A,X,Y,N).
state([P1,4],T1,T2',A',X,Y,N) :- inc(T2,T1,T2'),
    A'=<A,Y,A[Y-1]+A[Y-2]>, state([P1,3],T1,T2,A,X,Y,N).
state([P1,2],T1,T2',A,X,Y+2,N):-
    inc(T2,T1,T2'),state([P1,4],T1,T2,A,X,Y,N).

inc(T1,T2,T1') :- T1<=T2, T1+95<=T1'<=T1+105.
```

### 3.5 Microarchitecture

```
<0> j := 1
<1> while (j < 3) do
<2>   if (a[j] > a[j+1]) then <3> swap (a[j], a[j+1])
<4>   j := j + 1
end <5>
```

- We model timing characteristics due to microarchitecture considerations.
- Fixed assignment of cache line to instructions (direct-mapped cache).
- We assume the architecture has 2 cache lines: line 0 and 1, with each line contains at most 2 instructions.
- Instructions labeled with program points  $\langle 0 \rangle$ ,  $\langle 2 \rangle$  and  $\langle 4 \rangle$  are mapped to cache line 0, while  $\langle 1 \rangle$  and  $\langle 3 \rangle$  to cache line 1. A cache hit costs 1 time unit, while a miss costs 5 time units.

### CLP representation (bottom-up):

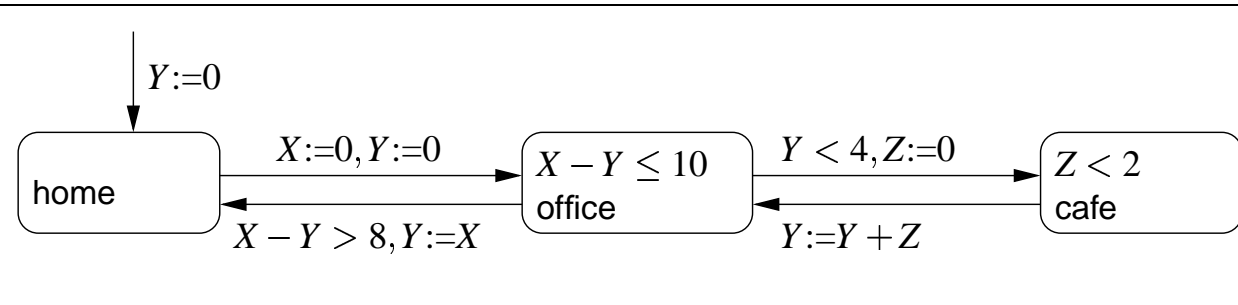
```
state(0,A,K,J,T,Tt) :- K = [[],[ ]],
    update(0,K,K1,E), state(1,A,K1,1,T+E,Tt).
state(1,A,K,J,T,Tt) :- J<3, update(1,K,K1,E),
    state(2,A,K1,J,T+E,Tt).
state(1,A,K,J,T,Tt) :- J>=3, update(1,K,K1,E),
    state(5,A,K1,J,T+E,Tt).
state(2,A,K,J,T,Tt) :- A[J]>A[J+1],
    update(2,K,K1,E), state(3,A1,K1,J,T+E,Tt).
state(2,A,K,J,T,Tt) :- A[J]<=A[J+1],
    update(2,K,K1,E), state(4,A,K1,J,T+E,Tt).
state(3,A,K,J,T,Tt) :- swap(A,J,J+1,A1),
    update(3,K,K1,E), state(4,A,K1,J,T+E,Tt).
state(4,A,K,J,T,Tt) :- update(4,K,K1,E),
    state(1,A,K1,J+1,T+E,Tt).
state(5,A,K,J,T,T).

update(Instr,[CL0,CL1],[CL0,CL1],1) :- in(Instr,CL0),!.
update(Instr,[CL0,CL1],[CL0,CL1],1) :- in(Instr,CL1),!.
update(Instr,[CL0,CL1],[CL01,CL1],5) :-
    cl_assgn(Instr,0), update_line(CL0,Instr,CL01).
update(Instr,[CL0,CL1],[CL0,CL11],5) :-
    cl_assgn(Instr,1), update_line(CL1,Instr,CL11).

update_line([],Instr,[Instr]). % cache empty
update_line([H1],Instr,[H1,Instr]). % partial
update_line([_,H2],Instr,[H2,Instr]). % cache full
```

- The variables  $K$  and  $K'$  represent the cache configuration: a pair of lists (one for each cache line), and each list contains at most two instructions.
- To verify that the worst-case execution time is 30, we can show that  $?- \text{state}(5, A, K, J, T, Tt)$  implies  $Tt \leq 30$ .

### 3.6 Non-Standard Timed Automaton

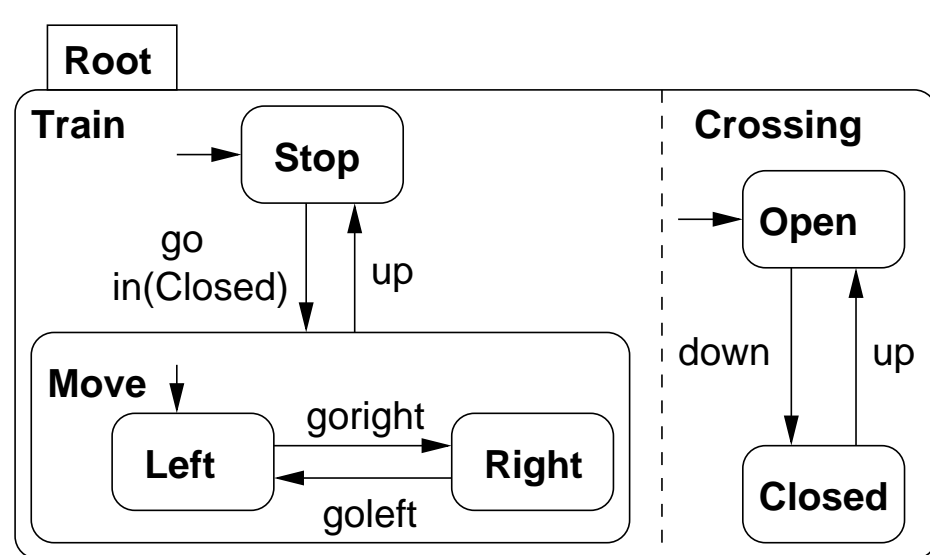


- A timed automaton describing a daily schedule of a worker.
- The variable  $Y$  is not a clock, but a simple continuous (real or rational) variable.
- Constraints involving both clocks and dynamically changing variables, as shown, cannot be handled by current timed automata model checkers.
- A worker cannot be out of the house for more than 20 time units can be shown by  $?- \text{state}(0, X, Y, Z)$  implies  $Y \leq 20$ . We can verify this automatically!
- More detail on timed automata modeling can be found in [6].

#### CLP representation (top-down):

```
state(home, X, Y, Z) :- E ≥ 0, X=E, Y=0, Z=E.
state(office, X1, Y1, Z1) :- E ≥ 0, X1=E, Y1=0, Z1=Z+E,
  X1-Y1 ≤ 10, state(home, _, Y, Z).
state(cafe, X1, Y1, Z1) :- E ≥ 0, X1=X+E, Y1=Y, Z1=E,
  Y < 4, X-Y ≤ 10, Z1 < 2, state(office, X, Y, _).
state(office, X1, Y1, Z1) :- E ≥ 0, X1=X+E, Y1=Y+Z, Z1=Z+E,
  Z < 2, X1-Y1 ≤ 10, state(cafe, X, Y, Z).
state(home, X1, Y1, Z1) :- E ≥ 0, X1=X+E, Y1=X, Z1=Z+E,
  8 < X-Y, X-Y ≤ 10, state(office, X, Y, Z).
```

### 3.7 Statecharts



#### CLP representation (bottom-up):

```
state(C, Ct) :- sctrans(C, go, C, C1), state(C1, Cf).
state(C, Ct) :- sctrans(C, up, C, C1), state(C1, Ct).
state(C, Ct) :- sctrans(C, down, C, C1), state(C1, Ct).
state(C, Ct) :- sctrans(C, go_right, C, C1), state(C1, Ct).
state(C, Ct) :- sctrans(C, go_left, C, C1), state(C1, Ct).
state(C, C).

sctrans(C, E, s(A, L), B) :- sctrans_def(C, E, s(A, L), B), !.
sctrans(C, E, s(A, L1), s(A, L2)) :- sctrans_and(C, E, L1, L2).

sctrans_and(_, _, [], []).
sctrans_and(C, E, [X|R], [Y|S]) :- sctrans(C, E, X, Y),
  sctrans_and(C, E, R, S).

sctrans_def(C, go, s(stop, []), s(move, [s(left, [])])) :-
  in(C, s(closed, [])).
sctrans_def(C, up, s(move, [_]), s(stop, [])).
...
```

- A Statechart model of a train crossing Statechart example from [1], modeled in CLP using the Statemate semantics [5].
- `sctrans/4` implements a recursion through the hierarchical structure of the statechart according to Statemate semantics: It finds a highest state in the hierarchy where an event is enabled, and changes the configuration according to the execution of the event defined in `sctrans_def/4` (we show two sample rules).
- Flexible modeling: Had we swapped the occurrence order of both rules, we obtain the UML semantics, which executes an event as low in the hierarchy as possible.

- Also extendible with history states.
- ‘The train is not in the state move while the crossing in the state open’ obtained by showing  $?- \text{state}(init, C), \text{in}(C, s(\text{move}, _)), \text{in}(C, s(\text{open}, _))$ , where  $init$  is the initial configuration, has no solution.

## 4 The Proof Method

### 4.1 Safety Assertions

- $G \models \Psi$ , where  $G$  is a goal and  $\Psi$  is a constraint.
- **Example:** Mutual exclusion for top-down representation of 2-process bakery algorithm:

```
state([P1, P2], T1, T2) ⊨ P1 ≠ 2, P2 ≠ 2.
```

- **Example:** In timed automata example, the worker cannot be out of the house for more than 20 time units:

```
state(home, X, Y, Z) ⊨ Y ≤ 20.
```

### 4.2 Outline

- Given a safety assertion  $A$  of the form  $G \models \Psi$  where  $G$  is a goal and  $\Psi$  a constraint, we perform unfolding toward the objective that each derived goal  $G'$  is either
  - *directly provable*, ie of the form  $p(\tilde{X}), \Psi_1 \models \Psi_2$  where  $\Psi_1 \models \Psi_2$  can be directly validated (and typically but not always, this is done when  $G'$  is terminal); or
  - *subsumed*, ie  $G'$  is an instance of another goal in another derivation sequence, or
  - *coinductive*, ie  $G'$  can be proved using the assumption that some parent goal is true.
- The second and third above requires a form of tabling, which we name *coinductive tabling*.
- The proof method is *compositional*, ie a proof of a program fragment (or procedure) can be directly used in the proof of the larger program.
- The proof method is based on the concept of unfold in CLP:
  - Let  $G = (B_1, \dots, B_n, \phi)$  and  $P$  denote a goal and program respectively. Let  $R = A : -C_1, \dots, C_m, \phi_1$  denote a rule in  $P$ , written so that none of its variables appear in  $G$ . Let  $A = B$ , where  $A$  and  $B$  are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of  $G$  using  $R$  is of the form
 
$$(B_1, \dots, B_{i-1}, C_1, \dots, C_m, B_{i+1}, \dots, B_n, B_i = A \wedge \phi \wedge \phi_1)$$
 provided  $B_i = A \wedge \phi \wedge \phi_1$  is satisfiable.
  - Given a program  $P$  and a goal  $G$  which contains one atom. Then  $\text{unfold}(G)$  is the set of reducts obtained by using *all* the rules.

### 4.3 Proof Rules

$$\begin{array}{l}
 \text{(UN)} \quad \frac{\Pi \cup \{\tilde{A} \vdash G \models \Psi\}}{\Pi \cup \bigcup_{i=1}^n \{\tilde{A} \cup \{G_i \models \Psi_i\} \vdash G_i \models \Psi\}} \quad \text{unfold}(G) = \{G_1, \dots, G_n\} \\
 \text{(AP)} \quad \frac{\Pi \cup \{\tilde{A} \cup \{G_1 \models \Psi_1\} \vdash G \models \Psi\}}{\Pi} \quad \text{there exists a substitution } \theta \text{ s.t. } \tilde{\forall}(G \models G_1\theta) \text{ and } \tilde{\forall}(\Psi_1\theta \models \Psi) \\
 \text{(AS)} \quad \frac{\Pi \cup \{\tilde{A} \cup \{G_1 \models \Psi_1\} \vdash G \models \Psi\}}{\Pi \cup \{\tilde{A} \cup \{G_1, \Psi_2 \models \Psi_1 \wedge \Psi_3\} \vdash G \models \Psi\}} \quad \Psi_2 \models \Psi_3 \text{ holds} \\
 \text{(DP)} \quad \frac{\Pi \cup \{\tilde{A} \vdash p(\dots), \Psi_1 \models \Psi\}}{\Pi} \quad \Psi_1 \models \Psi \text{ holds} \\
 \text{(SPL)} \quad \frac{\Pi \cup \{\tilde{A} \vdash G \models \Psi\}}{\Pi \cup \bigcup_{i=1}^k \{\tilde{A} \vdash G, \phi_i \models \Psi\}} \quad \phi_1 \vee \dots \vee \phi_k \text{ holds}
 \end{array}$$

- A *proof obligation* is of the form  $\tilde{A} \vdash G \models \Psi$ , where  $G$  is a goal,  $\Psi$  a constraint, and  $\tilde{A}$  is a set of assertions, called the *assumed* assertions.
- Each rule operates on the (possibly empty) set of proof obligations  $\Pi$ , by selecting a proof obligation from  $\Pi$  and attempting to discard it. In this process, new proof obligations may be produced.

- *Unfold* (UN) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from  $\Pi$ , is added into the set of assumed assertions of every newly produced proof obligation. Newly added assumed assertion may be used later in the application of rule (AP).
- *Assumption proof* (AP) directly proves an obligation by assuming the truth of an assumed assertion previously created by the rule (UN). Realizes the coinduction principle.
- *Direct proof* (DP), on the other hand, discards a proof obligation  $p(\dots), \Psi_1 \models \Psi$  directly, if the entailment could be proven.
- *Assumption specialization* (AS) specializes an assumption by adding constraints on both sides of the implication.
- *Split* (SPL) converts a proof obligation into several, more specialized ones.
- A safety assertion  $G \models \Psi$  holds if, starting with the proof obligation  $\Pi = \{\emptyset \vdash G \models \Psi\}$ , there exists a sequence of applications of proof rules that results in  $\Pi = \emptyset$ .

#### 4.4 Proving Simple CLP Program

```
p(0).
p(X + 2) :- p(X).
```

Assertion to prove:  $p(X) \models \text{even}(X)$ .

$$\begin{array}{c}
 \text{(UN)} \quad \frac{p(X) \models \text{even}(X) \quad \text{call this assertion } A}{\{A\} \vdash X=0 \models \text{even}(X)} \\
 \text{(DP)} \quad \frac{\{A\} \vdash X=0 \models \text{even}(X)}{\emptyset} \\
 \text{(AS)} \quad \frac{\{A\} \vdash p(X'), X=X'+2 \models \text{even}(X)}{\{p(V), W=V+2 \models \text{even}(V) \wedge W=V+2\} \vdash p(X'), X=X'+2 \models \text{even}(X)} \\
 \text{(AP)} \quad \frac{\{p(V), W=V+2 \models \text{even}(V) \wedge W=V+2\} \vdash p(X'), X=X'+2 \models \text{even}(X)}{\emptyset} \quad \text{even}(V) \wedge W=V+2 \models \text{even}(W)
 \end{array}$$

#### 4.5 Prototype Implementation

- A prototype implementation has been written purely in CLP( $\mathcal{R}$ ).
- The implementation uses a form of tabling called *coinductive tabling* to store assumed assertions.
- The difference to tabling in logic programming (e.g., XSB): Our system does not store redundant calls and answers in the table. Instead, it stores proof obligations.

### 5 Ongoing Work

#### 5.1 Abstraction

- The proof method described above can be augmented so as to implement the concepts of abstract interpretation and/or inductive assertions.
- Essentially, idea: any goal  $G$  in a derivation sequence may be *abstracted* by replacing the constraint  $\Psi$  in  $G$  with a more general one  $\Psi_1$ , that is,  $\Psi \models \Psi_1$ .
- A proof obligation  $G' \models \Psi$  is correct implies that  $G \models \Psi$ , where  $G'$  is an abstraction of  $G$ .

For example the assertion  $\text{state}(0,0,99,It,Nt) \models It=99$  is to be proven on the following bottom-up model.

```
state(0,I,N,I,N) :- I=0, N>=0.
state(1,I,N,It,Nt) :- I<N, state(0,I,N,It,Nt).
state(1,I,N,It,Nt) :- I<N, state(2,I,N,It,Nt).
state(2,I+1,N,It,Nt) :- state(1,I,N,It,Nt).
state(3,I,N,It,Nt) :- I>=N, state(0,I,N,It,Nt).
state(3,I,N,It,Nt) :- I>=N, state(2,I,N,It,Nt).
```

To use the loop-invariant  $i \leq n$  at program point 2, we abstract every goal in the proof of the form  $\text{state}(2,I,N,It,Nt), I = c, \dots$ , where  $c$  is a constant, into the form  $\text{state}(2,I,N,It,Nt), I \leq N, \dots$ . In doing so, the proof is obtained by analyzing the loop body once.

#### 5.2 Liveness

- Proving liveness for infinite-state problems requires *well-founded* induction.
- First identify a program point  $p$  and a well-founded measure  $m$  on program variables  $\tilde{X}$ . Then, when a goal of the form  $(\text{state}(p, \tilde{X}), \Psi)$  is encountered, split the proof process into two:

- $\text{state}(p, \tilde{X}), \Psi, m(\tilde{X}) = 0$
- $\text{state}(p, \tilde{X}), \Psi, m(\tilde{X}) > 0$

Prove the first, the base case, directly. Then prove the general case by assuming, in the proof process, that  $m(\tilde{X}) = c$  for some symbolic value  $c$ , and that liveness holds for  $\text{state}(p, \tilde{X}), \Psi, m(\tilde{X}) > 0$  when its proof subtree is covered by  $\text{state}(p, \tilde{X}'), \Psi'$ , where  $0 \leq m(\tilde{X}') < c$ .

- We prove the liveness of  $?- \text{state}(3,I,N,It,Nt), It=Nt$  in the above bottom-up model as follows:

1. We first perform an abstraction at program point 2.
2. For program point 2, we also provide a measure  $m(I,N,It,Nt) = N-I$ .
3. Further unfolding from  $\langle 2 \rangle$  to  $\langle 3 \rangle$  corresponds to the case  $m(I,N,It,Nt) = 0$ , since  $I \leq N$  (abstraction) and  $I \geq N$  (from loop condition) holds.
4. In another unfolding from  $\langle 2 \rangle$  to  $\langle 2 \rangle$ , we have a new value  $I'$  replacing  $I$ , and  $I' = I+1$  holds. Here, the measure  $N-I'$  is less than that of its parent, and therefore the proof concludes.

### 6 Related Work

- Work based on the XSB and XMC systems [7] used assertions based on the  $\mu$ -calculus and executed the logic program representations of programs and assertions using a tabling mechanism.
- Delzanno and Podelski [2] showed that a transition system and its CTL-based verification conditions can be translated into a CLP program in way that the symbolic CLP fixpoint operations can be used in the proof process.
- Works using CLP to describe program behavior, eg. that by Flanagan [3] for imperative programs, and that by Gupta and Pontelli [4] for timed automata.
- And many more.

### Acknowledgments

We thank the anonymous reviewers, and also Jinsong Dong for the non-standard timed automaton example.

### References

- [1] G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaard, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusability and compositionality. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 163–177. Springer, 1999.
- [2] G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. STTT*, 3(3):250–270, 2001.
- [3] C. Flanagan. Automatic software model checking using CLP. In P. Degano, editor, *12th ESOP*, volume 2618 of *LNCS*, pages 189–203. Springer, 2003.
- [4] G. Gupta and E. Pontelli. A constraint-based approach for specification and verification of real-time systems. In *18th RTSS*, pages 230–239. IEEE Computer Society Press, 1997.
- [5] D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM TOSEM*, 5(4):293–333, October 1996.
- [6] J. Jaffar, A. Santosa, and R. Voicu. A CLP proof method for timed automata. In *25th RTSS*, pages 175–186. IEEE Computer Society Press, 2004.
- [7] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In O. Grumberg, editor, *9th CAV*, volume 1254 of *LNCS*, pages 143–154. Springer, 1997.