# Modeling Systems in CLP with Coinductive Tabling

JOXAN JAFFAR, ANDREW E. SANTOSA, AND RĂZVAN VOICU

School of Computing
National University of Singapore
{joxan,andrews,razvan}@comp.nus.edu.sg

**Abstract.** We present a methodology for the modelling of complex program behavior in CLP. The first part of this paper is an informal description about how to represent a system in CLP. At its basic level, this representation captures the precise 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 of this paper 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 is proof 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 Introduction

Constraint Logic Programming (CLP) [11] has been successful as a framework for executable specifications, and several CLP systems have been successfully deployed widely in application areas such as artificial intelligence and combinatorial optimization (see eg. [14]). Its competitive advantage is expressive power coupled with a powerful inference method. In this paper, we present a systematic application of CLP to reasoning about complex *program behavior*.

We start with an informal but broad coverage of how to model various aspects of the operation semantics of programs. Starting with the basic concepts of strongest-postcondition and weakest-precondition, we then consider sequential and concurrent systems, including perpetual processes, and various synchronization mechanisms. Our modeling of these operational concepts is *exact* in the sense that the trace semantics of the underlying program is represented in the CLP model. Going further, we show how also to model the *machine* in which the program executes, focusing in particular, on the scheduler for nondeterministic actions (such as the choice of which process to execute next) and on the microarchitecture, which is often critical in determining the cost of execution. We thus cover reasoning not only about the values of program variables, but also about resources. At the end of this section, we show that a safety proof of a program

is established by executing a particular goal against the CLP model of the program, thus providing a generic and compositional methodology for program reasoning with CLP. However, basic CLP is not quite sufficient for this purpose.

The second part of this paper contains the main technical contribution: extending basic CLP with *coinductive tabling*. 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. This technique is akin to the notion of tabling in logic programming systems (see eg. [18]) in that the main purpose is to obtain termination when dealing with recursion. In standard tabling, procedure calls and their answers are tabled so as not to repeat them. In our tabling, the main differences are first that the setting is CLP and not just logic programming, and more importantly, that *proof obligations* and not just calls are tabled. (We do not table answers.) Termination is obtained by applying a principle of *coinduction* (see eg: Appendix B of [15]), that is: a recursive proof obligation may be proved by assuming that a preceding proof obligation is true.

Finally, we briefly discuss some important extensions that are not covered in this paper. The most important extension concerns the use of *abstraction*, in the manner of abstract interpretation [4]. We shall just explain that our proof method may be augmented by a liberally applying a notion of abstraction to a goal in a proof, preserving correctness. The advantage of abstraction is, of course, to enhance the process of termination. Another important extension is to deal with liveness or progress. Here, we discuss briefly how to include a notion of *well-founded induction*. Finally, we mention how to deal with an unbounded number of processes, ie: parameterized systems.

### 1.1 Related Work on CLP and Program Reasoning

There has been some recent work on using logic programs to describe concurrent programs, and which employ a systematic algorithm. Work based on the XSB and XMC systems [17] used assertions based on the $\mu$-calculus and executed the logic program representations of programs and assertions using a tabling mechanism. Delzanno and Podelski [6] 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. There are other works using CLP to describe program behavior, eg. that by Flanagan [8] for imperative programs, and that by Gupta and Pontelli [9] for timed automata, but these do not describe a systematic CLP-based algorithm.

## 2 CLP Preliminaries

We first briefly overview CLP [11]. The *universe of discourse* is a set of terms, integers, and arrays of integers. A *constraint* is written using a language of functions and relations. They are used in two ways, in the base programming language to describe expressions and conditionals, and in user assertions, defined below. In this paper, we will not define the constraint language explicitly, but invent them on demand in accordance

with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language.

An *atom*, is as usual, of the form $p(\tilde{t})$ where $p$ is a user-defined predicate symbol and the $\tilde{t}$ a tuple of terms. A *rule* is of the form $A$:-$\tilde{B},\phi$ where the atom $A$ is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint $\phi$ constitute the *body* of the rule. A *goal* $G$ has exactly the same format as the body of a rule. We say that a rule is a (constrained) *fact* if $\tilde{B}$ is the empty sequence. A *ground instance* of a constraint, atom and rule is defined in the obvious way.

Let $G = (B_1, \cdots, B_n, \phi)$ and $P$ denote a goal and program respectively. Let $R = A$:-$C_1, \cdots, C_m, \phi_1$ 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

$(B_1, \cdots, B_{i-1}, C_1, \cdots, C_m, B_{i+1}, \cdots, B_n, B_i = A \wedge \phi \wedge \phi_1)$

provided $B_i = A \wedge \phi \wedge \phi_1$ is satisfiable.

A *derivation sequence* is a possibly infinite sequence of goals $G_0, G_1, \cdots$ where $G_i, i > 0$ is a reduct of $G_{i-1}$. If there is a last goal $G_n$ with no atoms, notationally $(\Box, \phi)$ and called a *terminal goal*, we say that the derivation is *successful* and that the *answer constraint* is $\phi$. A derivation is ground if every reduction therein is ground.

In what follows, we shall only be concerned with goals that contain at most *one* atom. Thus the reduct of a goal is deterministic on the rule that is employed.

**Definition 1 (Unfold).** *Given a program $P$ and a goal $G$ which contains one atom. Then unfold$(G)$ is the set of reducts obtained by using* all *the rules. An* unfold operation *on $G$ results in a subset of unfold$(G)$; we say the operation is* complete *if it returns all of unfold$(G)$. An* unfold tree *of $G$ (sometimes also called a* proof tree*) is a tree of goals obtained by successively applying unfold operations on $G$, and then on the results of the previous operation, etc. We say that the unfold tree is* complete *if all the unfold operations used in its construction were complete*[1]. □

## 3 Modeling in CLP

We start by modeling the program $P$ with variables $\tilde{X}$ as a *predicate transformer* [7] by first identifying *target variables* $\tilde{X^t}$ corresponding to $\tilde{X}$, and then establishing a constraint on $\tilde{X}$ and $\tilde{X^t}$.

### 3.1 The Logical Basis of CLP Modeling

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 `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 `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. There are two alternatives

[1] Thus a complete tree is not necessarily one where unfold operations have been exhaustively performed; rather, it is where each unfold operation used was complete.

for implementing such a relation: *bottom up*, using strongest postcondition propagation, and *top-down*, using weakest precondition propagation. We introduce these concepts by example.

Consider the simple program `<0> x := x + 1 <1>` 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 `?- 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 `?- state(1, X, Xt), X > 5`, would return `Xt > 4`.

This simple idea is reminiscent of Hoare-like specifications of programs, such as $\{$`x > 5`$\}$ `x := x + 1` $\{$`x > 6`$\}$ or $\{$`x > 4`$\}$ `x := x + 1` $\{$`x > 5`$\}$, using the above examples. However, our approach is more powerful, in the sense that it can represent properties of *nonterminating* programs. Consider for example the program `<0> while (true) x := x + 1 <1> endwhile <2>` 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 `?- state(0, X, Xt), X > 5`, for example, has as solutions: `Xt > N` for all `N > 6`.

Figure 1 shows a while program and both its top-down and bottom-up CLP models. The target program points are 3, for the bottom-up model, and 0 for the top-down model. We note that for top-down models, target variables are required only by the compositional nature of our modeling. While this is an important feature, it is not the topic of interest in many of our examples; thus, for clarity, we remove the target variables from such examples.

We note that we could enhance the CLP model of a program such that the target program point become a parameter of the `state` predicate. The major benefit of this approach is that the CLP model of the program at hand allows queries on multiple target program points. Moreover, it is rather obvious that such a CLP model could be automatically derived for any program. While this approach is of practical importance, and has a straightforward implementation, it would unnecessarily complicate our examples. To illustrate our points, we shall rather use specialized, manually derived CLP models in our examples.

### 3.2 Concurrency

Consider our next example user program shown in Figure 2, a two-process Bakery mutual exclusion algorithm. Note that the point $\langle 2 \rangle$ indicates the critical section. Initially, $t_1 = t_2 = 0$. The CLP model is shown in Figure 3. Note that we now use a *pair* of program points, instead of just one. Note also that we consider "blocking" concurrency

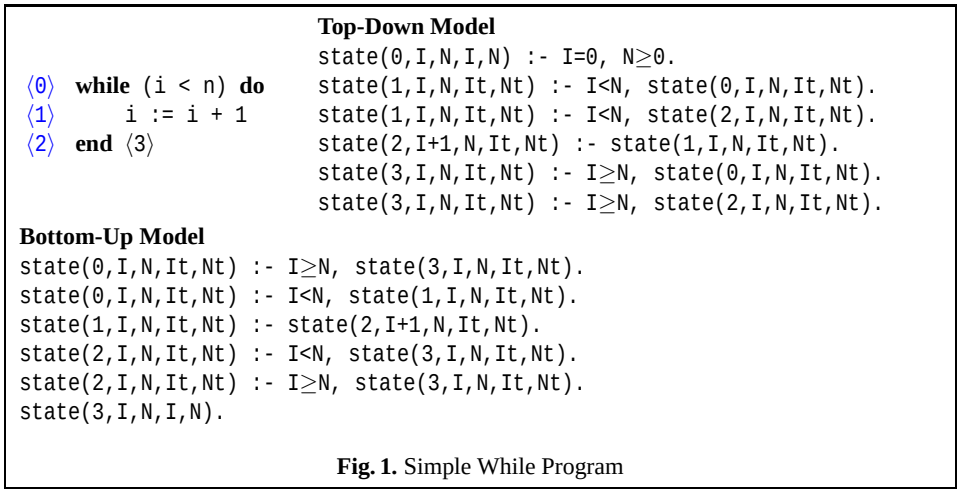


**Fig. 1.** Simple While Program

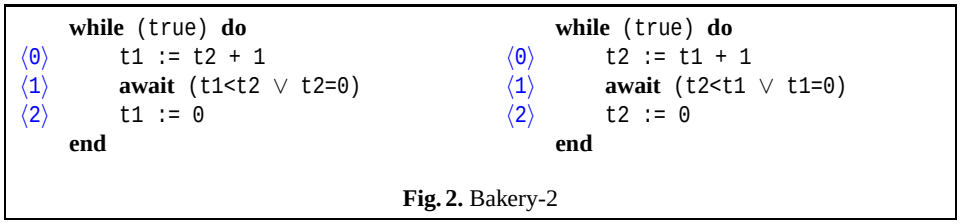


**Fig. 2.** Bakery-2

here, that is, we have an "**await**" statement, which is modelled to block until the specified condition holds. The safety property of interest, mutual exclusion, can be obtained by proving that `?- state([2,2], T1, T2)` has no solutions.

In this example, we adopted an asynchronous composition (instructions interleaving) of processes. Our framework is also flexible enough to model synchronous composition, as exemplified in [13]. It is also possible to replace the **await**s with busy loops, obtaining non-blocking concurrent programs.

### 3.3 High-Level Specifications

Consider a timed automaton [1] in Figure 4 describing a daily schedule of a worker. In this example, 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. This is because the standard algorithms depend on an analysis of clock regions, which use a more restrictive class of real constraints. Here, we may query if a worker can be out of the house for more than 20 hours, for example, by showing that `?- state(0,X,Y,Z)` implies $Y \leq 20$.

We show our top-down CLP model in Figure 5. See [13] for more details and examples of CLP modeling of timed automata.

```
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).
```

**Fig. 3.** Bakery-2 Top-Down CLP Model

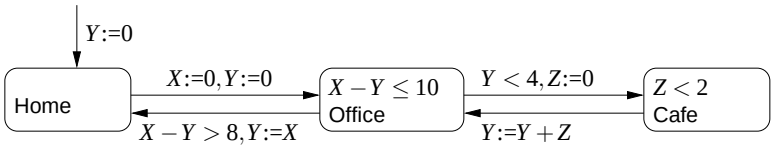


**Fig. 4.** Worker Timed Automaton

We now present an example of StateCharts. There are two main versions of their semantics: the Statemate semantics [10] and the UML semantics [16]. It is easy to adopt any of these for modeling in CLP. Here we show how we model a train crossing Statechart example of [3], using the Statemate semantics, in Figure 6.

We show our bottom-up CLP model in Figure 7. Here we model a state of a Statechart by a term `s(`*Name*`,` *Substates List*`)`. At any time, a *primitive state* has an empty substates list, an OR-state has only one state in its substates list, while an AND-state has more than one. The substates list is dynamically changing, depending on the currently active substate. Every Statechart has a conceptual *root* state, which is the topmost state in the hierarchy. We use the term *configuration* to denote a term of the form `s(root,` *Substates List*`)`, for example, the configuration

`s(root,[s(train,[s(stop,[])]),s(crossing,[s(closed,[])])])`

with which the execution of the Statechart begins. The state transition of a Statechart is thus a change from one configuration to another, which is triggered by some event[2].

Here we do not have program point variables (ie: there is just one point). The predicate `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`, for which we show two sample rules. Our modeling is flexible: 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. The modeling is also flexible enough to be extended with history states.

[2] Here we are ignoring the issue of step and superstep semantics, which is immaterial for the example at hand.

```
state(0,X,Y,Z) :- E≥0, X=E, Y=0, Z=E.
state(1,X1,Y1,Z1) :- E≥0,X1=E,Y1=0,Z1=Z+E,X1-Y≤10, state(0,_,Y,Z).
state(2,X1,Y1,Z1) :- E≥0,X1=X+E,Y1=Y,Z1=E,Y<4,X-Y≤10,Z1<2, state(1,X,Y,_).
state(1,X1,Y1,Z1) :- E≥0,X1=X+E,Y1=Y+Z,Z1=Z+E,Z<2,X1-Y1≤10, state(2,X,Y,Z).
state(0,X1,Y1,Z1) :- E≥0,X1=X+E,Y1=X,Z1=Z+E,8<X-Y,X-Y≤10,state(1,X,Y,Z).
```

**Fig. 5.** Worker CLP Model

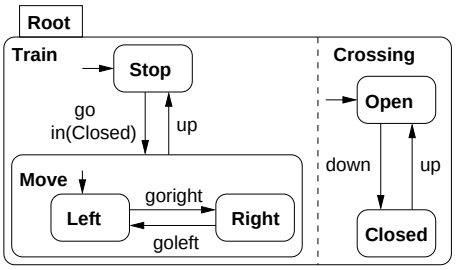


**Fig. 6.** Train Crossing Statechart

A property such as "the train is not in the state move while the crossing in the state open" can be obtained by showing that the `?- state(`*init*`,C), in(C,s(move,_)), in(C,s(open,_))`, where *init* is the initial confi guration, has no solution.

We can also potentially model other behavioral specifi cation such as Live Sequence Charts [5].

### 3.4 Scheduling

Here we model not just the visible aspect of the user program, but also the underlying scheduling mechanism. Such modeling is particularly important in the application domain of reasoning about resources, particularly time.

Consider the 2-process concurrent program shown in Figure 8. We wish to add the scheduling policy where Process 1 executes at least 1 and at most 3 instructions before control is passed to Process 2. Thus we implement "$k$-fairness" where $k = 3$ in this case.

We include this scheduling policy in our top-down CLP model shown in Figure 9, where we have a variable `Q` representing 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 `state([P1,P2],Q,X,Y)` implies $X \leq Y \times 3$.

Figure 10 shows a parallel Fibonacci program. The processes are run on separate processors, and access a shared array `a`. Initially `a[0] = 0`, `a[1] = 1`, and `a[`$i$`] = 0` for all $i \geq 2$. Process 1 assigns on the array `a`'s even indices $x$ the $x$-th Fibonacci number, while process 2 does the same with odd indices. Here the system performs no scheduling, but with the right timings, the program remains correct to an extent. We

```
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,[])).
...
```

**Fig. 7.** Train Crossing Bottom-Up CLP Model

```
     while (true) do                    while (true) do
⟨0⟩      x := x + 1            ⟨0⟩      y := y + 1
     end                                end
```

**Fig. 8.** Scheduled Concurrent Program

assume that every transition takes a fixed number $\varepsilon$ of time units, where $95 \leq \varepsilon \leq 105$. Under this assumption, when $\mathtt{N} \leq 3$, both process never access the same array location. That is, the goal `?- state([4,5],T1,T2,A,X,Y,N), N<=3` implies `A[N] = fib(N)`.

For $\mathtt{N} > 3$, however, computing $\mathtt{a}[i]$ could precede computing of $\mathtt{a}[i-1]$ for some $i$, that is, correctness is not guaranteed in case $\mathtt{N} > 3$.

The top-down CLP model is shown in Figure 11. Note that where $A$ is an array, we use the notation `A[I]` to denote the `I`-th element of `A`, and `<A,I,J>` to denote the array resulting from replacing the `I`-th element in `A` by `J`.

### 3.5 Microarchitecture

As in the subsection above, we seek here to model an internal component of the program's execution, in this case: timing characteristics due to microarchitecture considerations.

Consider a direct-mapped instruction cache architecture. Here, there is a fixed assignment of cache line to instructions. We assume the architecture has 2 cache lines: line 0 and 1, with each line contains at most 2 instructions. For the program in Figure 12, 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.

```
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).
```

**Fig. 9.** Top-Down CLP Model with Scheduling

```
⟨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⟩
```

**Fig. 10.** Dangerous Fibonacci with Fixed Timing

We implement these assumptions in our CLP model shown in Figure 13. The variables `K` and `K'` represent the cache confi guration: 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 `?- state(5,A,K,J,T,Tt)` implies `Tt` $\leq$ `30`.

Finally, it is also straightforward to model other architectural constraints such as data cache.

## 4 The Proof Method

Informally, the method is as follows. Given a *safety assertion* (or simply 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.

Note that 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.

### 4.1 The Logic

We now present a calculus for proving safety assertions. We start with

**Definition 2 (Proof Obligation).** *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.*

```
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,T21),
    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.
```

**Fig. 11.** Parallel Fibonacci Top-Down CLP Model

```
⟨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⟩
```

**Fig. 12.** Bubbling

Our proof rules are presented in Figure 14. 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. The proof process is typically centered around unfolding the goals in proof obligations.

The *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. Note that the resulting proof obligations are independent of one another. In each of the new proof obligations, the newly added assumed assertion may be used later in the application of rule (AP).

The rule *assumption proof* (AP) directly proves an obligation by assuming the truth of an assumed assertion previously created by the rule (UN). This rule therefore realizes the coinduction principle. The rule *direct proof* (DP), on the other hand, discards a proof obligation $p(\cdots), \Psi_1 \models \Psi$ directly, if the entailment could be proven. Both

```
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 line empty
update_line([H1],Instr,[H1,Instr]). % cache line not full
update_line([_,H2],Instr,[H2,Instr]). % cache line full
```

**Fig. 13.** Bubbling Bottom-Up CLP Model

(AP) and (DP) require an entailment proof by the constraint solver of CLP or a theorem prover. The rule *assumption specialization* (AS) specializes an assumption by adding constraints on both sides of the implication. Finally, the rule *split* (SPL) converts a proof obligation into several, more specialized ones. (This rule is not used in this paper, and is included for completeness.)

**Theorem 1 (Proof of Assertions).** *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 $\Pi = \emptyset$.* □

The proof rules above are sufficient in principle for our purposes. However, there is a very important principle with gives rise to an optimization: *redundancy* between goals.

The essential idea is this. Similar goals may be encountered in the unfold tree, each part of different proof obligations. For example, both obligation 1: $\tilde{A_1} \vdash G_1 \models \Psi$ and obligation 2: $\tilde{A_2} \vdash G_2 \models \Psi$, may be in $\Pi$, where there exists a renaming $\theta$ such that $\tilde{\forall}(G_2 \models G_1\theta)$. Suppose that further unfold of the goal of obligation 1 results in a proof subtree that does not coinductively use an assumed assertion in $\tilde{A_1}$. Hence it must be the case that the weaker obligation 2 is provable without $\tilde{A_2}$. However, proving obligation 2 is no longer necessary since obligation 1 is stronger. We exemplify this in the mutual exclusion proof of the Bakery algorithm in the next section.

$$(\text{UN})\quad \frac{\Pi \cup \{\tilde{A} \vdash G \models \Psi\}}{\Pi \cup \bigcup_{i=1}^{n} \{\tilde{A} \cup \{G \models \Psi\} \vdash G_i \models \Psi\}} \quad \mathit{unfold}(G) = \{G_1, \ldots, G_n\}$$

$$(\text{AP})\quad \frac{\Pi \cup \{\tilde{A} \cup \{G_1 \models \Psi_1\} \vdash G \models \Psi\}}{\Pi} \quad \begin{array}{l}\text{there exists a substitution } \theta \text{ s.t.} \\ \tilde{\forall}(G \models G_1\theta) \text{ and } \tilde{\forall}(\Psi_1\theta \models \Psi)\end{array}$$

$$(\text{AS})\quad \frac{\Pi \cup \{\tilde{A} \cup \{G_1 \models \Psi_1\} \vdash G \models \Psi\}}{\Pi \cup \{\tilde{A} \cup \{G_1, \Psi_2 \models \Psi_1 \wedge \Psi_3\} \vdash G \models \Psi\}} \quad \Psi_2 \models \Psi_3 \text{ holds}$$

$$(\text{DP})\quad \frac{\Pi \cup \{\tilde{A} \vdash p(\cdots), \Psi_1 \models \Psi\}}{\Pi} \quad \Psi_1 \models \Psi \text{ holds}$$

$$(\text{SPL})\quad \frac{\Pi \cup \{\tilde{A} \vdash G \models \Psi\}}{\Pi \cup \bigcup_{i=1}^{k} \{\tilde{A} \vdash G, \phi_i \models \Psi\}} \quad \phi_1 \vee \cdots \vee \phi_k \text{ holds}$$

**Fig. 14.** Proof rules

As noted above, the result of the application of (UN), each proof obligation represents an independent path of the unfold tree. Therefore in applying the rule, we have the freedom to unfold only partially first, and complete the unfold later. In our implementation (introduced in Section 6), we actually perform *depth-first* search. The CLP rule to be unfolded is also chosen based on its occurrence order in the CLP representation.

### 4.2 Example Proofs

For our first example, we explain coinduction. Consider the free-standing CLP program:

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

Consider the assertion $p(X) \models \mathit{even}(X)$, call it $A$, and its proof in Figure 15. The proof process starts by unfolding the $p(X)$ goal, resulting in two new proof obligations, each with the original goal $A$ as an *assumption*. On the left branch, after unfolding with the base-case clause, we are left with $X = 0 \models \mathit{even}(X)$, which can be discharged using the direct proof rule.

On the right branch of the proof, the unfolding rule produces the proof obligation $p(X'), X = X' + 2 \models \mathit{even}(X)$. Next we use the assumption specialization (AS) rule to modify the assumption $A$ as follows: (a) add $W = X + 2$ on both sides of $A$, where $W$ is new, and (b) rename $X$ into $V$, resulting in $p(V), W = V + 2 \models \mathit{even}(V) \wedge W = V + 2$.

Now consider applying the assumption proof rule (AP). We note that the side conditions (a) $p(X'), X = X' + 2 \models (p(V), W = V + 2)\theta$, and (b) $(\mathit{even}(V) \wedge W = V + 2)\theta \models \mathit{even}(X)$, where $\theta = \{V \mapsto X', W \mapsto X\}$, are true. Since all proof obligations have been discharged, the original assertion $p(X) \models \mathit{even}(X)$ is now proved.

We now consider the Bakery algorithm again, but now proceed informally. The mutual exclusion property of the Bakery algorithm of Section 3.2 is represented by the

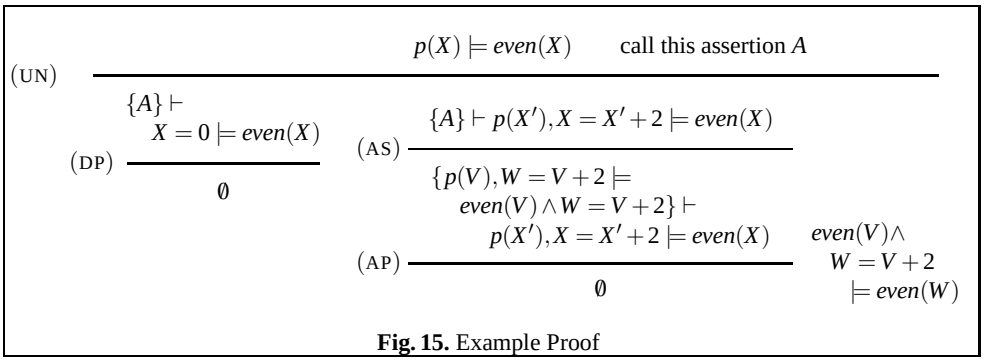


**Fig. 15.** Example Proof

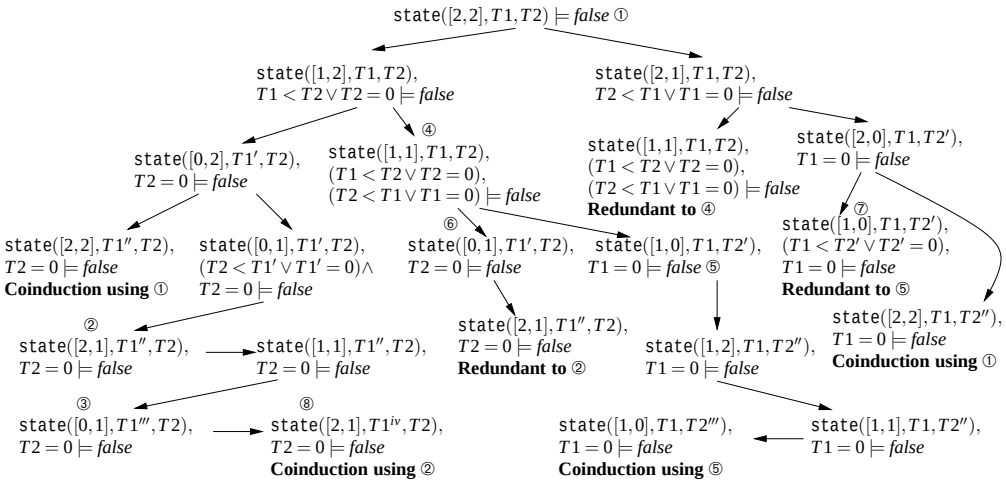


**Fig. 16.** Mutual Exclusion Proof for Bakery Algorithm

assertion `state([2,2],T1,T2)` $\models$ *false*. Figure 16 displays the proof sequence as the proof rules are applied on the original assertion above. We omit display of "successful" obligations in the picture where the premise has been reduced to *false*. The proof also includes a number of redundancy tests. Note that the obligation ⑥ is not considered redundant to obligation ③. This is because ③ contains in its assumed assertions the obligation ②, which is then used in its proof by application of coinduction at obligation ⑧. On the other hand, the redundancy of obligation ⑦ to ⑤ is valid, since none of the assumed assertions of ⑤ is used in its proof.

## 5 Beyond CLP

Here we briefly and informally discuss a few important extensions to the proof method, as our future work.

### 5.1 Abstraction

The use of abstract interpretation and/or inductive assertions is, in general, needed for the verification of infinite state problems (see eg. [2]). The proof method described above can be augmented so as to implement these concepts. Essentially, the idea is simply this: any goal in a derivation sequence may be *abstracted* by replacing the constraint $\Psi$ in $G$ with a more general one $\Psi_1$, that is, $\Psi \models \Psi_1$. This abstraction could be performed in accordance with the specification of an abstract domain, or in accordance to "loop-invariant" that is associated with the program point of $G$.

For example, recall the assertion `state(0,0,99,It,Nt)` $\models$ `It=99` to be proven on the bottom-up model of the while program shown in Figure 1. To use the loop-invariant $i \leq n$ at program point 2, we could abstract every goal in the proof of the form `state([2],I,N,It,Nt), I =` $c$, $\cdots$, where $c$ is a constant, into the form `state([2],I,N,If,Nf), I≤N,` $\cdots$. In doing so, the proof can be obtained without enduring the 99 steps.

Essentially, a proof obligation $G' \models \Psi$ is correct implies that $G \models \Psi$, where $G'$ is an abstraction of $G$, is also correct. Translating this simple but important observation into an effective addition to our proof method is a primary direction of our ongoing work.

### 5.2 Liveness

Intuitively, proving liveness or progress is a statement that from an initial goal $G$, all derivation sequences eventually reach a *target goal* $G^t$. Such a target goal could be defined, for example, as one with a particular value of the program counter. Checking liveness therefore is a matter of checking that there is a frontier of the proof tree in which *every* goal is a target goal.

As in the discussion about abstraction above, proving liveness for infinite-state problems would require a form of induction. In this case, what is required is *well-founded* induction, and we could proceed, informally, as follows. First identify a program point $p$ and a well-founded measure $m$ on program variables $\tilde{X}$. Then, when a goal of the form (`state(`$p$`,` $\tilde{X}$`),` $\Psi$`)` is encountered, split the proof process into two:

- `state(`$p$`,` $\tilde{X}$`),` $\Psi$, $m(\tilde{X}) = 0$
- `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 `state(`$p$`,` $\tilde{X}$`),` $\Psi$, $m(\tilde{X}) > 0$ when its proof subtree is covered by `state(`$p$`,` $\tilde{X}'$`),` $\Psi'$, where $0 \leq m(\tilde{X}') < c$.

Consider, for example, the program in Figure 1. To prove the liveness of `?- state(3,I,N,It,Nt), It=Nt`, we first need to perform an abstraction as explained in the previous section. In addition, for program point 2, we also provide a measure $m$(`I,N,It,Nt`) = `N-I`. Further unfolding from $\langle 2 \rangle$ to $\langle 3 \rangle$ corresponds to the case $m$(`I,N,It,Nt`) $= 0$, since `I≤N` (abstraction) and `I≥N` (from loop condition) holds. 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.

Automating this process, and to include the case where liveness/progress depends just on the *fairness* of the scheduler, is a major challenge.

### 5.3 Parameterized Systems

A parameterized system means that the number of processes is not fixed, but rather specified by a symbolic parameter. We choose to implement this idea as follows.

We first have a special Variable `Id`, which represents a static, unique process id to be referenced by its process code. The program counter shall also be an array `PC` of individual counters, such that `PC[Id]` represents the program counter of process `Id`. Program $P[Id]$ denotes the *infinite* system $P[1], P[2], \cdots$. As an example, we may define the parameterized program `P[Id] ::=` $\langle$`0`$\rangle$ **if** `(Id <` $n$`)` **then** `x := x + 1` $\langle$`1`$\rangle$.

We can represent the above parameterized program in CLP as follows:

```
state(Id,PC,X,N) :- PC[Id]=0, X=0.
state(Id,<PC,Id,1>,X+1,N) :- Id<N, state(Id,PC,X,N).
```

where `PC` is an array of program counters of length `N`.

We now explain some idea on how the parameterized system can be verified. Assuming there are `N` processes with `Id` ranging from 0 to `N`-1, for the above parameterized system we may want to prove the assertion `state(Id,PC,X,N), PC=[1,...,1]` $\models$ `X=N`, that is, at the end of execution `X=N`. The verification proceeds using induction on the number of processes (`N`). We first prove the base case for `N=1`, and the inductive case exemplified here: assuming `state(Id,PC,X,`$k$`), PC=[1,...,1]` $\models$ `X=`$k$ holds, we prove `state(Id,PC,X,`$k+1$`), PC=[1,...,1]` $\models$ `X=`$k+1$. From our hypothesis and other assumptions on the program, `state(Id,PC,X,`$k+1$`), PC=[1,...,1,0]` $\models$ `X=`$k$. We unfold `state(Id,PC,X,`$k+1$`), PC=[1,...,1]` $\models$ `X=`$k+1$, obtaining the goal `state(Id,PC',X',`$k+1$`), X'=X-1, PC'=[1,...,1,0]` $\models$ `X=`$k+1$. Using the hypothesis replacing `state(Id,PC',X',`$k+1$`)` with `X'=`$k$ the assertion is directly proven.

## 6 Experiments

W have implemented a prototype version of our proof method as a regular CLP($\mathcal{R}$) [12] program, with no special consideration for important features such as indexing to deal with the generally large number of generated assertions. The execution of our implementation is basically a call to `state` predicate. However, in addition we insert a tabling mechanism at every transition rule. Whenever a new proof obligation is generated by a call to the `state` predicate, the tabling mechanism will check whether the proof obligation is *subsumed* by another proof obligation already tabled. If so, it fails causing backtracking. Otherwise, it saves the new proof obligation in the table and continue.

The implementation uses a restricted form of coinduction, and verifies assertions only of the form `state(`$\tilde{X}$`),` $\phi \models$ *false*. A more general implementation of the future likely requires interfacing with a theorem prover.

## 7 Conclusion

We have presented a methodology for the modeling of complex program behavior in CLP. Our many examples demonstrated expressive elegance and power. We then presented a crucial extension to basic CLP, coinductive tabling, to facilitate execution of the model. We believe that our general approach, coupled with its potential for extension to deal with abstraction and liveness, can provide a signifi cant advance in the state of the art of program reasoning.

## References


1. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
2. K. R. Apt and E.-R. Olderog. *Verifi cation of Sequential and Concurrent Programs*. Graduate Texts in Computer Science. Springer, 2nd edition, 1997.
3. G. Behrmann, K. G. Larsen, H. R. Andersen, H. Hulgaar d, and J. Lind-Nielsen. Verification of hierarchical state/event systems using reusabi lity and compositionality. In R. Cleaveland, editor, *5th TACAS*, volume 1579 of *LNCS*, pages 163–177. Springer, 1999.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis. In *4th POPL*, pages 238–252. ACM Press, 1977.
5. W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *J. FMSD*, 19(1):45–80, 2001.
6. G. Delzanno and A. Podelski. Constraint-based deductive model checking. *Int. J. STTT*, 3(3):250–270, 2001.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall Series in Automatic Computation. Prentice-Hall, 1976.
8. C. Flanagan. Automatic software model checking using CLP. In P. Degano, editor, *12th ESOP*, volume 2618 of *LNCS*, pages 189–203. Springer, 2003.
9. 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.
10. D. Harel and A. Naamad. The STATEMATE semantics of Statecharts. *ACM TOSEM*, 5(4):293–333, October 1996.
11. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. LP*, 19/20:503–581, May/July 1994.
12. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS*, 14(3):339–395, 1992.
13. 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.
14. K. Marriott and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.
15. F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
16. Object Management Group, Inc. *OMG Unifi ed Modeling Language Specifi cation*, March 2003. Version 1.5 formal/03-03-01.
17. 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.
18. D. S. Warren. Memoing for logic programs. *Communications of the ACM*, 35(3):93–111, March 1992.