A CLP Proof Method for Timed Automata

Joxan Jaffar, Andrew Santosa and Răzvan Voicu
School of Computing, National University of Singapore

Abstract

Constraint Logic Programming (CLP) has been used to model programs and transition systems for the purpose of verification problems. In particular, it has been used to model Timed Safety Automata (TSA). In this paper, we start with a systematic translation of TSA into CLP. The main contribution is an expressive assertion language and a new CLP inference method for proving assertions. A distinction of the assertion language is that it can specify important properties beyond traditional safety properties. We highlight one important property: that a system of processes is symmetric. *The new inference mechanism is based upon the well-known method of tabling in logic programming. It is distinguished by its ability to use assertions that are not yet proven, using a principle of coinduction. Apart from given assertions, the proof mechanism can also prove implicit assertions such as discovering a lower or upper bound of a variable. Finally, we demonstrate significant improvements over state-of-the-art systems using standard TSA benchmark examples.*

1 Introduction

Constraint Logic Programming (CLP) [12] has been used to model programs and transition systems for the purpose of verification problems. In particular, it has been used to model Timed Safety Automata (TSA) [1, 9]. The main advantage of using CLP pertain to expressiveness. For example, [6] demonstrates the proof of some standard properties, as well as properties such as time bounds between important events, on a CLP representation of TSA. No systematic algorithm, however, was presented.

The standard algorithms for reasoning about TSA are based on time regions/zones and the generic method of model-checking; see eg. the recent survey [3] which includes a description of the Uppaal system. Using a tabling logic programming framework, the XMC/RT system [5, 16] showed how a TSA represented in CLP can be comparable in efficiency. Here the assertion language was based on the $\mu - calculus$, and the fixpoint operators therein were realized by the tabling mechanism without which termination could not be assured. Importantly, all the standard algorithms for verifying TSA are based on a *finite* representation, upon which their search-based algorithms depend for termination.

In this paper, we start with a systematic translation of TSA into CLP. The main contribution is an expressive assertion language and and a new CLP inference method for proving assertions. A distinction of the assertion language is that it can specify important properties beyond traditional safety properties. We highlight one important property: that a system of processes is *symmetric*. We argue that many other important properties can be similarly proved. A key advantage of proving such assertions, apart from their own truth, is to use them in the efficient proof of other assertions.

We present a new inference mechanism which is based upon the well-known method of tabling in logic programming. Our method is distinguished by its ability to use assertions that are not yet proven, using a principle of *coinduction*. Not only does coinduction allow (like normal induction) the consideration of infinite state systems, it does not require the discovery of a well-founded measure nor a base case, as is traditional. Apart from proving given assertions, our proof method can also prove *implicit* assertions such as discovering a lower or upper bound of a variable, often useful in real-time systems.

The use of coinduction and the use of proven assertions is then coupled with a (standard tabling) method of *redundancy*, which eliminates further computation where it is not needed. This coupling, we shall argue, is a significant implementation methodology for search-based algorithms which can deal with infinite state problems. We finally provide empirical evidence by demonstrating significant improvements over state-of-the-art systems using standard TSA benchmark examples

2 CLP modelling of Timed Safety Automata

2.1 CLP Preliminaries

We present some preliminary definitions about CLP [12].

The *universe of discourse* $\mathcal{D}$ of our CLP programs is a set of terms, integers, and lists 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. The set $\{p(\tilde{d})\}$ where p ranges over the predicates and $\tilde{d}$ ranges over the tuples in $\mathcal{D}$ is called the *domain base* $\mathcal{B}$ of our CLP programs. A *rule* is of the form $A\,{:}{-}\tilde{B},\Psi$ where the atom A is the *head* of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the *body* of the rule. A *goal* 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 is obtained by instantiating variables therein from $\mathcal{D}$. Such an instantiation is called a *grounding* of the constraint, and the result is *true* or *false*. Similarly, a *ground instance* of an atom or rule is obtained by a grounding which instantiates variables therein with values in $\mathcal{B}$. The *ground instances* of a goal G, written $[\![G]\!]$ is the set of ground atoms obtained by taking all the true ground instances of G and then assembling the ground atoms therein into a set. We write $G_1 \models G_2$ to mean that for all groundings θ of G_1 and G_2, each ground atom in $G_1\theta$ appears in $G_2\theta$.

Let $G = (B_1,\cdots,B_n,\Psi)$ and P denote a goal and program respectively. Let $R = A\,{:}{-}C_1,\cdots,C_m,\Psi_1$ denote a rule in P, written so that none of its variables appear in G. Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B. A *reduct* of G using R is of the form

$$(B_1,\cdots,B_{i-1},C_1,\cdots,C_m,B_{i+1},\cdots,B_n,B_i = A \wedge \Psi \wedge \Psi_1)$$

provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ has a true ground instance. The reduction is *ground* if G is ground, and the reduct above is replaced by one of its ground instances. An *atom selection* rule determined the choice of the atom B_i used for reduction above.

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 $(\square,\Psi)$ and called a *terminal goal*, we say that the derivation is *successful* and that the *answer constraint* is Ψ. A derivation is ground if every reduction therein is ground.

Finally, consider the fixpoint operator T_P for P which maps $\mathcal{D}$ into $\mathcal{D}$ as follows: a ground atom A is in $T(S)$ if there is a ground instance $A\,{:}{-}B_1,\cdots,B_n$ of a rule in P such that $\{B_1,\cdots,B_n\} \subseteq S$.

A basic theorem of CLP is that the least fixpoint of T_P is the least model of P, and this is also equal to the set of ground atoms, considered as a (solitary) goal, which have a successful ground derivation.

2.2 Constraint Traces for TSA

Timed safety automata (TSA) [9] have been proposed for the specification of real-time systems. In this subsection we shall provide a brief introduction to TSAs and their parallel composition as means of modelling real-time systems, and define the *constraint trace* of a TSA, which is a CLP-based semantics useful in reasoning about safety properties.

A TSA is a structure $\langle \Sigma, Q, q_0, \mathcal{C}, \mathcal{B}, \Delta, I\rangle$, where Σ is the input alphabet of *events*, or *actions*, Q is a finite set of *locations*, q_0 is the *initial location*, $\mathcal{C}$ is a finite set of *clock variables* that range over positive real numbers, $\mathcal{B}$ is a *language of constraints* over $\mathcal{C}$, $\Delta \subseteq Q \times \Sigma \times Q \times \mathcal{B} \times 2^{\mathcal{C}}$ is the *transition relation*, and $I : Q \mapsto \mathcal{B}$ is a mapping that associates a *location invariant* to every location. We shall typically denote elements of the sets Σ, Q, $\mathcal{C}$, and $\mathcal{B}$, by the following, possibly subscripted symbols: s, q, c, and B, respectively. We shall also denote sets of clock variables by the letter C. Given a transition $\langle q,s,q',B,\{c_1,\ldots,c_k\}\rangle$, q represents the current location, s the event/action that triggers the current transition, q' is the next location, B is a constraint over $\mathcal{C}$ that must hold when the transition occurs, and $\{c_1,\ldots,c_k\} \subseteq \mathcal{C}$ a set of clock variables to be reset during the current transition. Given a TSA, a *clock valuation* is a mapping from its set of clock variables $\mathcal{C}$ to positive real numbers. On the set of valuations we define the following partial order. Given two valuations $v_1,v_2 : \mathcal{C} \mapsto \mathbb{R}_+$, we say that $v_1 \preccurlyeq v_2$ if $v_1(c) \leq v_2(c)$ for every clock variable $c \in \mathcal{C}$. Given a set of clock variables $\mathcal{C} = \{c_1,\ldots,c_k\}$, we say that a clock valuation satisfies a clock constraint B if the result of the substitution $B[c_1/v(c_1),\ldots,c_k/v(c_k)]$ is a ground constraint that holds. For simplicity, we shall assume that the location invariants are convex, i.e., given a location invariant I and three clock valuations $v_1,v_2,v_3 : \mathcal{C} \mapsto \mathbb{R}_+$ such that $v_1 \preccurlyeq v_2 \preccurlyeq v_3$, we have that v_2 satisfies I whenever v_1 and v_3 satisfy I.

A TSA operates in the following way. The automaton starts off in its initial location q_0, with all the clock variables set to zero. The clock variables are incremented uniformly, and at the same rate, modelling the passage of time. The TSA takes as input an infinite timed word, that is, it responds to an infinite sequence of time-stamped events $(s_1,t_1)\cdots(s_k,t_k)\cdots$, where $(s_k,t_k) \in \Sigma \times \mathbb{R}_+$ and $t_{i+1} > t_i \geq 0$, for all $i > 0$. Assuming that the current location of the TSA is q, the occurrence of a time-stamped event (s,t) shall trigger a transition of the automaton into location q' if the following conditions hold:

a) There exists a transition $\langle q,s,q',B,\{c_1,\ldots,c_k\}\rangle \in \Delta$, where $k \geq 0$.

b) B is satisfied by the values of the clock variables at time t.

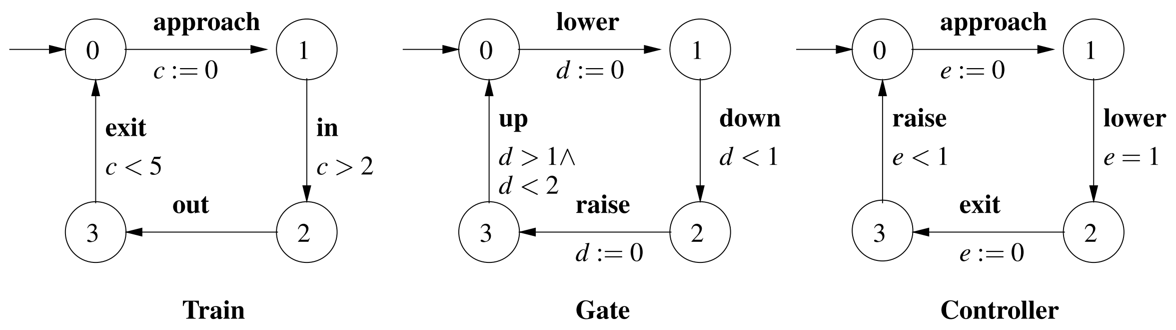


Figure 1. TSA specification of a train crossing

c) $I(q)$ is satisfied by the values of the clock variables for the entire duration of q being the current location of the TSA[1].

The transition to the new location q' is instantaneous, and resets the clock variables $c_1, \ldots, c_k$ to zero. If, upon the occurrence of a time-stamped event in the input sequence, no transition is possible, we say that the timed word $(s_1,t_1)\cdots(s_k,t_k)\cdots$ is *not accepted* by the TSA.

Several comments are now in order. The typical definition of TSAs (e.g. [3]) uses either a Büchi or Muller acceptance condition for a timed word; for the purpose of this paper, however, a simpler acceptance condition is sufficient. Therefore we shall only require that the timed word "stays within" the automaton indefinitely. Other works provide TSA definitions that limit the language of clock constraints $\mathcal{B}$ to constraints of the form $c \odot n$, or $c_1 \odot c_2$, where $\odot \in \{\leq, <, =, >, \geq\}$, and n is a positive integer. Also, [5] discusses the possibility of extending a TSA with *discrete variables*, i.e. non-clock variables taking values over finite domains. Such limitations ensure the termination of the verification procedure, or its efficient execution. As it shall be shown in subsequent sections, our work employs inductive reasoning and tabling to achieve a finite verification procedure; for this reason, we do not need to limit the clock constraint language, and can also accomodate non-clock variables that are not required to be discrete.

We continue with the definition of the parallel composition of timed safety automata. Given a set of TSAs $\{T_1, \ldots, T_n\}$, where each T_k has the form $\langle \Sigma_k, Q_k, q_0^k, \mathcal{C}_k, \mathcal{B}_k, \Delta_k, I_k \rangle$, for all k, $1 \leq k \leq n$, with $Q_{k_1} \cap Q_{k_2} = \emptyset$, for all $k_1 \neq k_2$ and $1 \leq k_1, k_2 \leq n$, the *parallel composition* of $T_1, \ldots, T_n$ is the TSA $\langle \bigcup_{1\leq k\leq n} \Sigma_k, \prod_{1\leq k\leq n} Q_k, \langle q_0^1, \ldots, q_0^n \rangle, \bigcup_{1\leq k\leq n} \mathcal{C}_k, B, \Delta, I \rangle$, where

- $\mathcal{B} = \{\bigwedge_{1\leq k\leq n} B_k \,|\, B_k \in \mathcal{B}_k\}$;
- $\Delta = \{\langle \langle q_1, \ldots, q_n \rangle, s, \langle q_1', \ldots, q_n' \rangle, \bigwedge_{1\leq k\leq n} B_k, \bigcup_{1\leq k\leq n} C_k \rangle \,|\,$ if $s \in \Sigma_k$ then $\langle q_k, s, q_k', B_k, C_k \rangle \in \Delta_k$ else $q_k = q_k', B_k = true, C_k = \emptyset\}$;
- $I : \prod_{1\leq k\leq n} Q_k \mapsto \mathcal{B}$, $I(\langle q_1, \ldots, q_k \rangle) = \bigwedge_{1\leq k\leq n} I_k(q_k)$.

[1] In fact, it is sufficient to check that $I(q)$ is satisfied at the beginning and the end of the transition, due to the convexity of I.

Denote by T the parallel composition of the timed safety automata $T_1, \ldots, T_n$. Intuitively, each T_i runs independently inside T, responding only to events in Σ_i. Since the alphabets Σ_i, $1 \leq i \leq n$ are not necessarily disjoint, it is often the case that an event triggers synchronous transitions in more than one TSA; such synchronous transitions are the means by which the TSAs communicate with one another. Given a time-stamped event (s,t), all TSAs whose alphabet contains s must perform a transition upon the occurrence of this event, or the current timed word shall not be accepted.

Figure 1 shows a train crossing specified as the parallel composition of three TSAs: the train, the gate and the controller. Figure 2 shows the TSA representing the parallel composition of the three TSAs in Figure 1. We note that the events **approach** and **exit** are in the alphabets of both the train and the controller, while the events **lower** and **raise** are in the alphabets of both the gate and the controller. All these four symbols trigger synchronous transitions; for example, the transition from location $[1,0,1]$ to location $[1,1,2]$ on event **lower** in Figure 2 represents synchronous transitions of both the gate and the controller.

We now introduce a CLP-based semantics of timed safety automata. Consider a TSA $T = \langle \Sigma, Q, q_0, \{c_1, \ldots, c_n\}, \mathcal{B}, \Delta, I \rangle$, $n \geq 0$. Given a transition $\tau = \langle q, s, q', B, C_{reset} \rangle \in \Delta$, we denote by $trans_\tau$ the CLP clause

$$\texttt{trans}(q, q', [C_1, \ldots, C_n], [C_1', \ldots, C_n'], Interval, s) \\ to\_clp(B), C_1' = C_1'' + Interval, \ldots, C_n' = C_n'' + Interval$$

where $C_1, \ldots, C_n$ and $C_1', \ldots, C_n'$ are CLP variables that represent the values of the clock variables $c_1, \ldots, c_n$ at the point when the TSA transitions *out of* locations q and q', respectively, *Interval* represents the duration between the two transitions, and $to\_clp(B)$ represents the translation into CLP of the constraint B, such that the clock variables $c_1, \ldots, c_n$ are replaced by the CLP variables $C_1, \ldots, C_n$. Moreover,

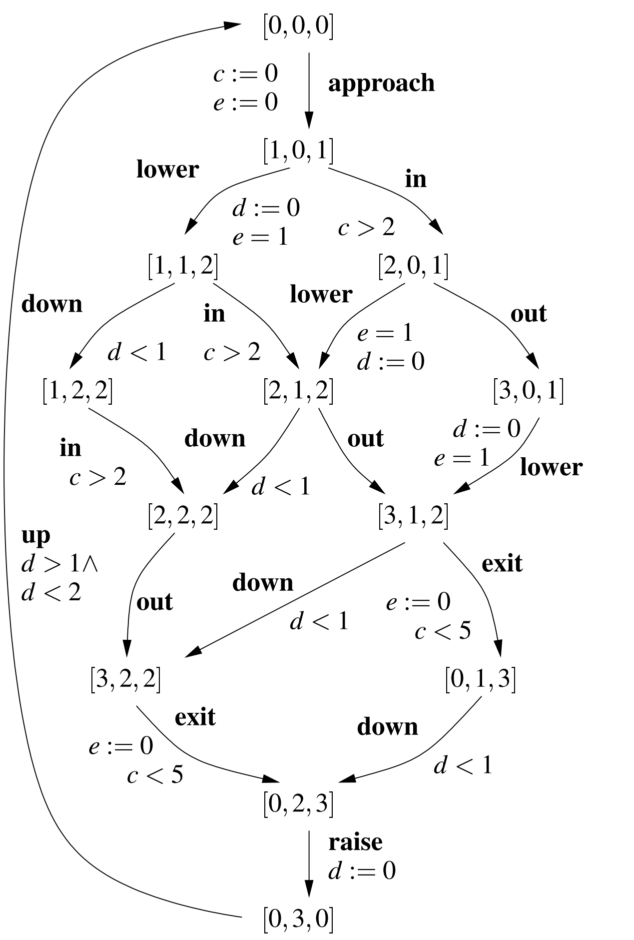


Figure 2. TSA Parallel Composition

the symbols C_i'', $1 \leq i \leq n$ are placeholders for either 0, if $c_i \in C_{reset}$, or C_i, if $c_i \notin C_{reset}$. We also define the set $Trans_T$ as $\{trans_\tau \,|\, \tau \in \Delta\}$. Next, given a location invariant $I(q)$, for some $q \in Q$, we denote by inv_q the CLP clause

$$\texttt{inv}(q, [C_1, \ldots, C_n]) :- to\_clp(I(q))$$

and by Inv_T the set of clauses $\{inv_q \,|\, q \in Q\}$. Finally, we introduce the predicate cs, whose solution space represents all the reachable configurations $\langle q, v, t \rangle$, where, q is a location of T, v is a clock valuation over $\{c_1, \ldots, c_n\}$, and t is the point in time when the automaton *transitions out of q*. To facilitate the process of reasoning about T, the predicate cs has a trace of observables as one of its arguments. The definition of cs is given by the following two CLP clauses:

$\texttt{cs}(q_0, [Int, \ldots, Int], 0, [])$
$\texttt{cs}(Q, Clocks, T, [(S, T) \,|\, Rest]) :-$
$\quad \texttt{inv}(Q, Clocks),$
$\quad \texttt{trans}(QPrev, Q, ClocksPrev, Clocks, Interval, S),$
$\quad \texttt{trans}(QPrev, Q, ClocksPrev, ClocksTmp, 0, S),$
$\quad \texttt{inv}(Q, ClocksTmp), Interval \geq 0,$
$\quad \texttt{cs}(QPrev, ClocksPrev, T - Interval, Rest)$

where the length of the list $[Int, \ldots, Int]$ is n. Intuitively, the first argument represents the current state, *clocks* represent the list of clock variables, T represents the current time, and the last argument represents a list of timestamped events pertaining to the trace.

We denote the least model of the two clauses above by CS_T.

Definition 1 (Constraint Trace) *Given a time safety automaton $T = \langle \Sigma, Q, q_0, \mathcal{C}, \mathcal{B}, \Delta, I \rangle$, the* constraint trace $\mathcal{CT}_T$ *of T is the set of clauses $Trans_T \cup Inv_T \cup CS_T$.*

Given a TSA $T = \langle \Sigma, Q, q_0, \{c_1, \ldots, c_n\}, \mathcal{B}, \Delta, I \rangle$, consider a a quadruple $\langle q, v, t, w \rangle$. It is rather straightforward to prove that w is the reversed prefix of an acceptable timed word that takes T through a sequence of transitions from its initial location, to the location $q \in Q$, at time t, with the clock valuation v, if and only if $\texttt{cs}(q, [v(c_1), \ldots, v(c_n)], t, w)$ is in the success set of predicate cs.

Figure 3 shows a simplified version of the constraint trace of the train crossing automaton given in Figure 2. The train crossing automaton has $I(q) = true$ for all locations q in the three automata. For this reason, any call to the inv predicate succeeds, and for simplicity, we have partially evaluated the CLP program in Figure 3 and removed the inv predicate completely.

Consider the safety condition that between two approaches there is a **down** before every **in**. We can verify this by running the following CLP goal, and obtaining *failure*. Let $\Psi(L)$ be a constraint such that the list L has a prefix L_0 which (a) contains an **in** and only later in the list, a **down**, and (b) does not contain any **approach**. Then we could run the following goal, and expect failure:

$$?-\ cs(\_, \_, \_, [(approach, \_) | Rest]), \Psi(Rest)$$

Finally, consider proving that between every **approach** and **exit** there will pass at most 5 seconds. Let $\Psi(L, T2)$ be a constraint such that the list L has a prefix L_0 which contains no occurrence of **approach** and **exit** except at the end, and the last element in L_0 is $(approach, T2)$. Then we could run the following goal, and expect failure:

$$?-\ cs(\_, \_, \_, [(exit, T1) | Rest]), \Psi(Rest, T2), T2 - T1 > 5.$$

3 Assertions

Definition 2 (Assertion) *An* assertion *is of the form $G \models G'$ where G is a nonterminal goal and G' is a possibly terminal goal.*

```
trans([0,0,0],[1,0,1],[C,D,E],[C1,D1,E1],T,approach):-
            C1 = T, D1 = D+T, E1 = T .
trans([1,0,1],[1,1,2],[C,D,E],[C1,D1,E1],T,lower):-
            E = 1, C = C1+T, D1 = T, E1 = E+T .
trans([1,0,1],[2,0,1],[C,D,E],[C1,D1,E1],T,in):-
            C>2, C1 = C+T, D1 = D+T, E1 = E+T .
trans([1,1,2],[1,2,2],[C,D,E],[C1,D1,E1],T,down):-
            D<1, C1 = C+T, D1 = D+T, E1 = E+T .
trans([1,1,2],[2,1,2],[C,D,E],[C1,D1,E1],T,in):-
            C>2, C1 = C+T, D1 = D+T, E1 = E+T .
trans([2,0,1],[2,1,2],[C,D,E],[C1,D1,E1],T,lower):-
            E = 1, C1 = C+T, D1 = T, E1 = E+T .
trans([2,0,1],[3,0,1],[C,D,E],[C1,D1,E1],T,out):-
            C1 = C + T, D1 = D+T, E1 = E+T .
trans([1,2,2],[2,2,2],[C,D,E],[C1,D1,E1],T,in):-
            C>2, C1 = C+T, D1 = D+T, E1 = E+T .
trans([2,1,2],[2,2,2],[C,D,E],[C1,D1,E1],T,down):-
            D<1, C1 = C+T, D1 = D+T, E1 = E+T .
trans([2,1,2],[3,1,2],[C,D,E],[C1,D1,E1],T,out):-
            C1 = C+T, D1 = D+T, E1 = E+T .
trans([3,0,1],[3,1,2],[C,D,E],[C1,D1,E1],T,lower):-
            E = 1, C1 = C+T, D1 = T, E1 = E+T .
trans([2,2,2],[3,2,2],[C,D,E],[C1,D1,E1],T,out):-
            C1 = C+T, D1 = D+T, E1 = E+T .
trans([3,1,2],[3,2,2],[C,D,E],[C1,D1,E1],T,down):-
            D<1, C1 = C+T, D1 = D+T, E1 = E+T .
trans([3,1,2],[0,1,3],[C,D,E],[C1,D1,E1],T,exit):-
            C<5, C1 = C+T, D1 = D+T, E1 = T .
trans([3,2,2],[0,2,3],[C,D,E],[C1,D1,E1],T,exit):-
            C<5, C1 = C+T, D1 = D+T, E1 = T .
trans([0,1,3],[0,2,3],[C,D,E],[C1,D1,E1],T,down):-
            D<1, C1 = C+T, D1 = D+T, E1 = E+T .
trans([0,2,3],[0,3,0],[C,D,E],[C1,D1,E1],T,raise):-
            C1 = C+T, D1 = T, E1 = E+T .
trans([0,3,0],[0,0,0],[C,D,E],[C1,D1,E1],T,up):-
            D>1, D<2, C1 = C+T, D1 = D+T, E1 = E+T .

cs([0,0,0],[T,T,T],0,[]) .
cs(Q,Clocks,T,[(S,T)|Rest) :-
            trans(QPrev,Q,ClocksPrev,Clocks,T1,S),
            T1 ≥ 0, cs(QPrev,ClocksPrev,T-T1,Rest).
```

Figure 3. Constraint Trace Example

This assertion is neither just a safety assertion, nor just a liveness assertion. It can be be used for both purposes, but it has yet a unique interpretation.

First, it is clearly usable as a safety assertion by having G' as a terminal goal. For example, returning to the train example described in section 2.2, the safety condition could be expressed as the assertion:

$$cs(\_,\_,\_,[(approach,\_)|Rest]),\Psi(Rest) \models false$$

Now, the assertion is also usable to specify a kind of liveness property, or more precisely, a reachability property. An assertion of the form

$$cs(\tilde{X}) \models cs(\tilde{Y}),\Psi$$

specifies that if there is an execution trace to a state $cs(\tilde{X})$, then there is also a trace to a state of the form $cs(\tilde{Y}),\Psi$.

It is important to note, however, that our assertion does not state *temporal* or trace-based properties, that is, it does not say that a specific state *follows* from another.

We now present a key application of the assertion: the specification of *symmetry* between processes. Intuitively, a symmetry between goals is such that a valuation of a goal can be transformed into an equivalent valuation of the symmetric goal.

Consider for example two processes, P_1, which contains a distinguished variable X_1, and similarly, P_2 with X_2. Symmetry may express that the runtime behavior of P_1 wrt X_1 is isomorphic to that of P_2 wrt X_2, eg. for every value of X_1 observed at a program point i in P_1, so too is that value observed in X_2 at program point i in P_2. The extension of this notion to more processes and/or more variables, is intuitively clear (though complicated to formalize).

A typical kind of symmetry of interest in this paper (but not the only kind that our general method can handle) is as follows. Suppose we have $n > 0$ processes $P_1, P_2, \cdots, P_n$ and there are n (proper) variables. Let ρ denote a permutation of $1, 2, \cdots, n$. To say that the list $(P_1, P_2, \cdots, P_n)$, $n > 0$, of processes is symmetric wrt to the n variables, we shall use an assertion of the form:

$$\begin{array}{l} cs([A_1,\cdots,A_n],[X_1,\cdots,X_n]) \models \\ \quad cs([A_{\rho(1)},\cdots,A_{\rho(n)}],[X_{\rho(1)},\cdots,X_{\rho(n)}]) \end{array}$$

whose simplest instantiation is:

$$cs([A_1,A_2],[X_1,X_2]) \models cs([A_2,A_1],[X_2,X_1])$$

which states that the the runtime behavior of P_1 wrt X_1 is isomorphic to that of P_2 wrt X_2, A more complicated example is the Fischer mutual exclusion example in Figure 7. An example symmetry assertion we use, for the two-process Fischer example below, is

$$cs([P_0,P_1],[X_0,X_1,K]) \models cs([P_1,P_0],[X_1,X_0,K']),\Psi$$

where the constraint Ψ constrains K to be in $\{-1,0,1\}$ and constrains K' as follows:

$$K' = \begin{cases} -1 & \text{if } K = -1 \\ 0 & \text{if } K = 1 \\ 1 & \text{if } K = 0 \end{cases}$$

and we shall illustrate a proof below. Adapting such symmetry assertions to more than two processes is straightforward.

Note the important fact that symmetry is neither a safety nor a liveness kind of property. That is, symmetry does not describe a superset of states that covers a goal, nor does it describe that any particular state eventually follows from another state. In fact, our general assertion is able to state *structural* properties about processes. Symmetry is used in this paper as a driving example. In general, there are many other kinds of properties that may be usefully specified.

In the literature, a well known approach to symmetry is [11] who provided a sybtactic condition for a class of programs to be symmetric. Our approach, in contrast, is semantic. Further, the assertions we use as examples of symmetry properties are just that: examples. In general, our assertions can specify a larger class of non-behavioral properties.

In this paper, we shall consider just one other application of the assertion: (plain) safety. More specifically, these are assertions in the form

$G \models \Psi$

(or, equivalently, $G, \neg\Psi \models false$). In the experiments section below, we shall consider two standard benchmark classes of programs, the Fischer benchmark for proving mutual exclusion within a fixed number of processes, and a bridge crossing benchmark for proving that a fixed number of processes representing trains can safely cross a bridge. In each case, we shall first prove symmetry assertions, and then, using these as confirmed assertions, prove the safety assertions.

4 CLP and Coinductive Tabling

Here we present an algorithm schema for proving assertions. The essential idea is to start with the premise of the assertion at hand as a goal G, and incrementally construct a standard proof tree for G. Initially, the assertion is labelled as an "assumed" assertion, and is the only one. As the proof tree is constructed, it is generally the case that more assumed assertions emerge, and some assumed assertions become proved. These new assumed assertions are each associated with a distinct node in the tree. We finally prove the original assertion when the terminal nodes, interpreted as a disjunction of goals, and using in some way the collection of (assumed and proved) assertions generated, imply the conclusion of the assertion.

The key contribution is a method of limiting the size of the proof tree. Firstly, we need to obtain a finite size, and secondly, a "small" size. The method is a unification of two concepts: *coinduction* and *redundancy*, and is capable of dealing with *infinite* state problems. We next elaborate and formalize this.

4.1 Proof Trees and Assertion Tables

A *proof tree* for a goal G is a tree of goals rooted at G. Each node G in the the tree may be associated with an assertion where G is the premise of the assertion. Each such assertion is indicated as either *assumed* or *confirmed*. The assertions associated with a proof tree, together with a number of confirmed assertions (probably arising from previous proofs), are collectively known as the *assertion table* of the tree. Initially, there is one assumed assertion to be confirmed, and the proof tree is simply the one node labelled with the premise of this assertion.

A proof tree (and its assertion table) is incrementally constructed, initially from the tree which contains just G, by a number of proof tree extensions. Each such extension is characterized by an operation on a leaf goal in the tree which transforms this leaf goal into a tree. The result is a generally bigger proof tree for G. These operations perform one of the following: (a) transforms a leaf goal in a proof tree into a (sub)tree of goals, (b) add a new assumed assertion to the assertion table, and (c) reclassify an assumed assertion into a confirmed one. The formal definition is now given.

Definition 3 (Unfold) *Let G be a leaf node in a proof tree, associated with an assumed assertion $G \models G'$. An* unfold *operation on G results in a proof tree consisting of G at the root, and* all *the reducts of G, say G_i, $1 \leq i \leq n$, as descendant nodes. Further, we now have a new assumed assertion $G_i \models G'$ associated to the descendant node G_i of G, for each $1 \leq i \leq n$.* □

In the transformation steps below, before a (confirmed or assumed) assertion is chosen to be used, the assertion is first renamed so that all the variables therein are fresh.

Let a *goal renaming* be a set of variable-variable equations. We shall consider such a set as a substitution θ on terms, and write $G\theta$ to mean the goal obtained from G by renaming the variables therein according to θ, and we assume that the goals G and $G\theta$ have no common variables.

Definition 4 (Applying Coinduction) *Let G be a leaf node in a proof tree, associated with the assertion $G \models G'$. Suppose there is a parent node associated with an assumed assertion $G_0 \models G_1$. Also suppose that there is a goal renaming θ and a constraint Ψ, whose variables are limited to those in G_0 and fresh variables, such that $G \models G_0\theta \wedge \Psi$. Then we may apply* coinduction *on G using this assertion, and the result is to replace G by a new node $G_1\theta \wedge \Psi$. The associated assertion of this new node is $G_1\theta \wedge \Psi \models G'$.* □

A key point here is that the use of assumed assertions, *though unproven*, can be used to confirm an assertion. Intuitively, when considering a node in the proof tree, using an assumed assertion of a parent node is tantamount to the use of "coinduction" (see eg. Appendix B of [15]). One therefore need not discover a well-founded measure (such as that in the Hoare rule for termination [10]), nor a base case.

The next step, applying a confirmed assertion, is almost identical, but a little simpler.

Definition 5 (Applying a Confirmed Assertion) *Let G be a leaf node in a proof tree, associated with the assertion $G \models G'$, and let $G_0 \models G_1$ be some confirmed assertion. Suppose that there is a goal renaming θ and a constraint Ψ, whose variables are limited to those in G_0 and fresh variables, such that $G \models G_0\theta \wedge \Psi$. Then we may replace G by a new node $G_1\theta \wedge \Psi$. The associated assertion of this new node is $G_1\theta \wedge \Psi \models G'$.* □

Note that the above two steps may produce a new goal G which is terminal. In such cases, further unfolding is not possible, and the associated assertion associated with the node will have to be proved directly.

The next transformation step is seemingly similar to the above two steps, but is in fact typically used as a complement.

Definition 6 (Applying Redundancy) *Let G be a leaf node in a proof tree, associated with the assertion $G \models G'$, and let there be another node associated with an assumed assertion $G_0 \models G_1$. Suppose that there is a goal renaming θ and a constraint Ψ, whose variables are limited to those in G_0 and fresh variables, such that $G \models G_0\theta \wedge \Psi$ and $G_1\theta \wedge \Psi \models G'$. Then we may apply* redundancy *on G by simply replacing both G and its assocaited assertion with the terminal goal true.* □

The use of redundancy here is key in obtaining a finite proof tree, by avoiding recomputations, for example, loops. In this regard, there is similarity with the standard tabling method for logic programs (see eg. [4]) wherein a repeated call to a predicate is delayed. However, standard tabling serves to compute answers, and terminate when the set of answers is complete. In this sense, it is a "bottom-up" computation. In our case, the use of tabling is for detecting when coinduction can be used, and also when a goal is redundant. In this sense, our method is "top-down".

The process of confirming assertions is discussed next.

4.2 Proving Assertions

Let G be a node in a proof tree, and $G \models G'$ be the associated assertion. We say that G is *directly proved* if one of the following holds.

1. both G and G' are terminal, ie they are both just constraints, and the constraint entailment $G \models G'$ can be established by the underlying constraint solver;

2. G is of the form $cs(\tilde{X_1}), \Psi_1$ and G' is of the form $cs(\tilde{X_2}), \Psi_2$, and the constraint entailment $\tilde{X_1} = \tilde{X_2}, \Psi 1 \models \Psi_2$ can be established by the underlying constraint solver;

3. there is a reduct G'' of G', and there is a direct proof of $G \models G''$.

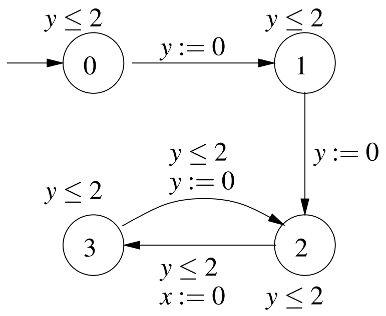


Figure 4. Clock Boundedness Example

Only the third and recursive case 3 needs some explanation. In principle, the implementation of this case is challenging, for there is generally a large space of possible reducts of the goal G'. In practice, however, it is often sufficient to limit the possibilities in a simple way. For example, it is in fact often sufficient not to consider any reducts at all (ie. the recursive case is not needed), and sometime, it is sufficient just to consider *immediate* reducts of G'.

We now come to the main result, that a proof tree confirms the assumed assertion associated to its root node if all of its frontier nodes can be directly proved.

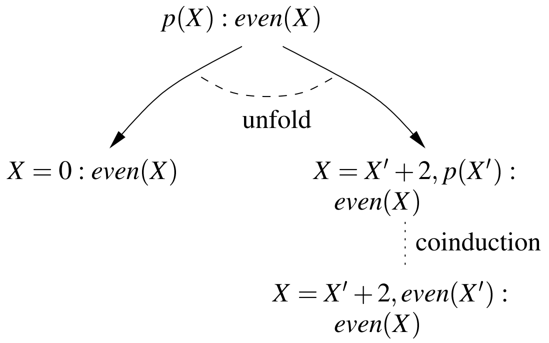
Theorem 1 (Proving an Assertion)
Let G be the root of a proof tree. Then its associated assumed assertion is proved if all nodes in the frontier of the tree can be directly proved. □

For example, consider the free-standing CLP program:

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

Consider the proof tree below whose root node $p(X)$ has the assumed assertion $p(X) \models even(X)$, where *even* is a unary constraint with the obvious meaning. Note that we depict the node $p(X)$ and its associated assertion in the form $p(X) : even(X)$.

Performing on unfold step produces two more nodes, the first of which is directly proved: $X = 0 \models even(X)$. Consider the second node $X = X' + 2, p(X')$ and the goal renaming $X \mapsto X'$. Since $p(X') \models p(X)\theta$ and $even(X)\theta$ is $even(X')$, we can apply coinduction using the assumed assertion $p(X) \models even(X)$ in order to replace this node by $X = X' + 2, even(X')$. Finally, it is straightforward to directly prove that this (terminal) goal implies $even(X)$, and thus the node is directly proved. Since both nodes in the frontier have been directly proved, the original assertion $p(X) \models even(X)$ is now proved.



4.3 Proving Implicit Assertions

In addition to proving pre-specified assertions, the proof tree also contains *implicit* information. That is:

if each node in the frontier implies some goal G, then G holds.

In our considered application domain of TSA, a particularly important kind of implicit information is that of a *bound* for a variable. For example, it is useful (and sometimes even critical) to know if a certain clock has a lower or upper bound. That is, a worst or best case analysis of execution time is a key application.

We thus now consider implicit information of the form $X \geq \alpha$ or $X \leq \alpha$, where α denotes a number, contained in a proof tree for a goal G. By assuming the existence of a *projection* mechanism in the underlying constraint solver (and CLP($\mathcal{R}$) [13] has this feature), we can compute, for each goal in the frontier, a lower or upper bound for a particular variable, or determine that no lower or upper bound exists. These implicit bounds, made explicit by the projection mechanism of the constraint solver, thus can determine if there is a lower or upper bound for a variable as a precondition to the goal G.

Consider, for instance, the example in [14], shown in Figure 4. The idea here is to discover that there is an upper bound for the clock X, and further, its value (in this case, 6). Consider the goal $\texttt{cs}(Loc,[X,Y],\_,\_) \models X \leq \alpha$ on the program in Figure 5, where α is a constant left unspecified. The unfolding process shall bind the location variable *Loc* to each of the four locations of the automaton. For simplicity, we shall show the unfolding when *Loc* has already been bound to 2. The proof tree is shown in Figure 6. The left branch of the tree unfolds to location 3, and then again to location 2, leading to redundant computation. By using Definition 6, we can apply redundancy and replace the $\texttt{cs}(2,[X2,Y2],\_,\_)$ goal by *true*. The right branch unfolds to location 1, and then to location 0, where the unfolding stops. By projecting on variable X on both branches, we get the constraints $X \leq 2$ from the left branch, and $X \leq 6$ from the right branch, and thus we can infer that $\alpha = 6$.

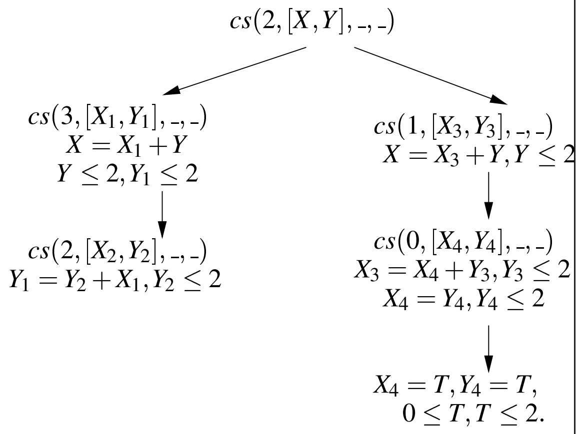


Figure 6. Proof Tree for Clock Boundedness Example

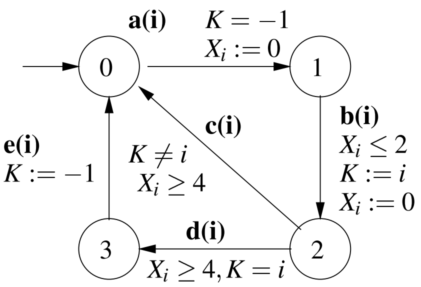


Figure 7. Fischer's Algorithm

The proof tree in Figure 6 shows that we can use an implicit assertion $G \models X \leq \alpha$, where α is a symbolic constant whose value shall be inferred later. Though, formally speaking, the proof is coinductive, we note that the constraint $X \leq \alpha$ did not play a role in the proof, due to the fact that both clock variabes X and Y are reset periodically in every run of the TSA. In general, this need not be the case. In the case of generalized TSAs, augmented with non-discrete variables, we may need to infer a constraint Ψ for α such that the coinductive step holds. Then, the upper bound for X shall be the minimal value of α such that Ψ holds.

4.4 An Example Proof

We now prove symmetry for the two-process version of the Fischer example. The constraint trace is shown in Figure 8. Recall that the symmetry assertion is

$$cs([P_0,P_1],[X_0,X_1,K]) \models cs([P_1,P_0],[X_1,X_0,K']),\Psi$$

```
cs(0,[T,T],T,[]) :- T≥0,T≤2.
cs(1,[X,Y],T,[(_ ,T)|Rest]) :- Y≤2, Y1≤2, X=X1+Interval, Y=Interval, Interval≥0, cs(0,[X1,Y1],T-Interval,Rest).
cs(2,[X,Y],T,[(_ ,T)|Rest]) :- Y≤2, Y1≤2, X=X1+Interval, Y=Interval, Interval≥0, cs(1,[X1,Y1],T-Interval,Rest).
cs(3,[X,Y],T,[(_ ,T)|Rest]) :- Y≤2, Y1≤2, X=Interval, Y=Y1+Interval, Interval≥0, cs(2,[X1,Y1],T-Interval,Rest).
cs(2,[X,Y],T,[(_ ,T)|Rest]) :- Y≤2, Y1≤2, X=X1+Interval, Y=Interval, Interval≥0, cs(3,[X1,Y1],T-Interval,Rest).
```

Figure 5. Constraint Trace for Boundedness

```
cs([0, 0], [X0, X1, -1]) :- Z ≥ 0, X0 = Z, X1 = Z. % init
cs([1, P1], [Y0, Y1, L]) :- K = -1, Z ≥ 0, Y0 = Z, Y1 = X1+Z, cs([0, P1], [X0, X1, K]). % a(0)
cs([2, P1], [Y0, Y1, L]) :- X0 ≤ 2, L = 0, Z ≥ 0, Y0 = Z, Y1 = X1+Z, cs([1, P1], [X0, X1, K]). % b(0)
cs([0, P1], [Y0, Y1, K]) :- X0 ≥ 4, K ≠ 0, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([2, P1], [X0, X1, K]). % c(0)
cs([3, P1], [Y0, Y1, K]) :- X0 ≥ 4, K = 0, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([2, P1], [X0, X1, K]). % d(0)
cs([0, P1], [Y0, Y1, L]) :- L = -1, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([3, P1], [X0, X1, K]). % e(0)
cs([P0, 1], [Y0, Y1, K]) :- K = -1, Z ≥ 0, Y0 = X0+Z, Y1 = Z, cs([P0, 0], [X0, X1, K]). % a(1)
cs([P0, 2], [Y0, Y1, L]) :- X1 ≤ 2, L = 1, Z ≥ 0, Y0 = X0+Z, Y1 = Z, cs([P0, 1], [X0, X1, K]). % b(1)
cs([P0, 0], [Y0, Y1, K]) :- X1 ≥ 4, K ≠ 1, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([P0, 2], [X0, X1, K]). % c(1)
cs([P0, 3], [Y0, Y1, K]) :- X1 ≥ 4, K = 1, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([P0, 2], [X0, X1, K]). % d(1)
cs([P0, 0], [Y0, Y1, L]) :- L = -1, Z ≥ 0, Y0 = X0+Z, Y1 = X1+Z, cs([P0, 3], [X0, X1, K]). % e(1)
```

Figure 8. Constraint Trace for Fischer's Algorithm

where the constraint Ψ constrains K to be in $\{-1,0,1\}$ and constrains K' as follows:

$$K' = \begin{cases} -1 & \text{if } K=-1 \\ 0 & \text{if } K=1 \\ 1 & \text{if } K=0 \end{cases}$$

This can be formally stated more simply as three assertions:

$$\begin{array}{l} cs([P_0,P_1],[X_0,X_1,-1]) \models cs([P_1,P_0],[X_1,X_0,-1]) \\ cs([P_0,P_1],[X_0,X_1,0]) \models cs([P_1,P_0],[X_1,X_0,1]) \\ cs([P_0,P_1],[X_0,X_1,1]) \models cs([P_1,P_0],[X_1,X_0,0]) \end{array}$$

We shall illustrate briefly the proof of just the first assertion.

Consider unfolding the premise $cs([P_0,P_1],[X_0,X_1,-1])$; we obtain seven nodes, but focus on just one, that obtained by using the CLP rule `a(0)`. This node, illustrated together with its assumed assertion, is

$$\begin{array}{l} cs([0,P_1],[X'_0,X'_1,-1]), P_0=1, X'_1 \le X_1, X_0 \le X_1 : \\ \quad cs([P_1,P_0],[X_1,X_0,-1]). \end{array}$$

Now apply coinduction to this and obtain

$$\begin{array}{l} cs([P_1,0],[X'_1,X'_0,-1]), P_0=1, X'_1 \le X_1, X_0 \le X_1 : \\ \quad cs([P_1,P_0],[X_1,X_0,-1]). \end{array} \tag{1}$$

Consider now the conclusion $cs([P_1,P_0],[X_1,X_0,-1])$ of the assertion. Using the rule `a(1)`, we obtain the reduct

$$cs([P_1,0],[X''_1,X''_0,-1]), P_0=1, X''_1 \le X_1, X_0 \le X1.$$

Call this G. Now, observe that the goal $cs([P_1,0],[X'_1,X'_0,-1]), P_0=1, X'_1 \le X_1, X_0 \le X_1$ in (1) implies G. By the (third) rule for direct proving above, this proves the assertion as desired.

Returning now to the other six nodes obtained in unfolding the premise of the assertion. These are obtained by using the CLP rules `c(0), e(0), a(1), c(1), e(1), init`. We informally outline these proofs for three cases `c(0), e(0), init`.

$cs([2,P_1],[X'_0,X'_1,-1])$ $\quad P_0=0, X'_0 \ge 4, X_0 \ge X'_0 :$ $\quad\quad cs([P_1,P_0],[X_1,X_0,-1])$	Unfold premise by `c(0)`.
$cs([P_1,2],[X'_1,X'_0,-1])$ $\quad P_0=0, X'_0 \ge 4, X_0 \ge X'_0 :$ $\quad\quad cs([P_1,P_0],[X_1,X_0,-1])$	Apply symmetry.
$cs([P_1,2],[X'_1,X'_0,-1])$ $\quad P_0=0, X'_0 \ge 4, X_0 \ge X'_0 \models$ $\quad\quad cs([P_1,2],[X''_1,X''_0,-1])$ $\quad\quad\quad P_0=0, X''_0 \ge 4, X_0 \ge X''_0$	Goal implies reduct of conclusion by `c(1)`.

$cs([3,P_1],[X'_0,X'_1,K'])$ $\quad P_0=0 :$ $\quad\quad cs([P_1,P_0],[X_1,X_0,-1])$	Unfold premise by `e(0)`.
$cs([P_1,3],[X'_1,X'_0,-1])$ $\quad P_0=0 :$ $\quad\quad cs([P_1,P_0],[X_1,X_0,-1])$	Apply symmetry.
$cs([P_1,3],[X'_1,X'_0,-1]),$ $\quad P_0=0 \models$ $\quad\quad cs([P_1,3],[X''_1,X''_0,-1])$ $\quad\quad\quad P_0=0$	Goal implies reduct of conclusion by `e(1)`.

The next three cases, for the rules `a(1), c(1), e(1)`, are similar and hence omitted. Finally, we deal with the `init` rule. Here the proof proceeds slightly differently. First, we constrain the variables of the premise using the rule `init` as follows:

$$cs([P_0,P_1],[X_0,X_1,-1]), P_0 = P_1 = 0, X_0 = X_1, X_0 \geq 0 : \\ cs([P_1,P_0],[X_1,X_0,-1]).$$

We then apply the symmetry assertion to the goal, getting

$$cs([P_1,P_0],[X_1,X_0,-1]), P_0 = P_1 = 0, X_0 = X_1, X_0 \geq 0 : \\ cs([P_1,P_0],[X_1,X_0,-1]).$$

As we can see, the goal immediately implies the conclusion.

5 Experimental Results

The purpose of this section is to demonstrate the feasibility and promise of the general technique described above. We consider both execution time as well as the size of the search space. For execution time, we demonstrate that the technique is feasible. We implemented a prototype as a normal CLP($\mathcal{R}$) [13] program. We thus did not implement important special features such as an indexing mechanism, which is important because assertion tables get very big. More importantly, we consider the search space in order to demonstrate scalability.

We run two standard benchmarks, and one program to demonstrate the usage of circular symmetry. We compare to two systems, a prototype of XMC/RT [16] and Uppaal Version 3.4.6 [2]. We chose the former because it is based on logic programming. It is implemented on top of XSB [17], and thus utilizes a custom tabling mechanism. We chose the latter because it is a state-of-the-art and well-known system for timed automata. We remark that these two systems utilize the technique of Difference Bound Matrices, thus limiting the class of constraints.

Because we are not comparing systems here, we omit comparison with other related systems such as HyTech [8], Kronos [20], RED [18, 19], etc. In all these systems, the use of data structures is central. While XMC/RT and Uppaal use DBM's, RED uses a BDD-like data-structure. These structures have demonstrated significant performance.

We ran our experiments on Pentium 4 Xeon cluster node with 2.0 GB RAM and minimum CPU clock speed set to 2.0 GHz.

Fischer's Algorithm. See Figure 7 where i ranges over the number of processes. We prove mutual exclusion, that no two processes can be at state 3. We assume symmetry for Fischer, and its general definition, where N is the number of processes, is as follows:

$$cs([P_0,\ldots,P_{N-1}],[X_0,\ldots,X_{N-1},-1]) \models \\ cs([P_{\rho(0)},\ldots,P_{\rho(N-1)}],[X_{\rho(0)},\ldots,X_{\rho(N-1)},-1]), \\ cs([P_0,\ldots,P_{N-1}],[X_0,\ldots,X_{N-1},K]), K \neq -1 \models \\ cs([P_{\rho(0)},\ldots,P_{\rho(N-1)}],[X_{\rho(0)},\ldots,X_{\rho(N-1)},\rho(K)]),$$

Bridge Crossing Problem. We adapted the bridge crossing problem for N trains, in Figure 9, from the Uppaal 3.4.6 package. The system consists of two kinds of processes: a controller and a number of trains. We use $2N+1$ variables: $X_1,\ldots,X_N$ are clocks for each train, the next N variables $Pos_0,\ldots,Pos_N$ represent the positions of each train in the queue, and the last variable Len stores the length of the train queue. From the original Uppaal model, we translate the sequence of transitions that visit committed locations between both endpoints into single transition without abstracting out the operations. The system's state space thus remains the same since committed locations are not part of Uppaal's state space. We verified that there is at most one train in the crossing (state 2) at any given time.

The symmetry definition that we used for the Bridge Crossing Problem is as follows:

$$cs([A_0,A_1,\ldots,A_N],[X_1,\ldots,X_N,Pos_1,\ldots Pos_N,Len]) \models \\ cs([A_0,A_{\rho(1)},\ldots,A_{\rho(N)}],[X_{\rho(1)},\ldots,X_{\rho(N)}, \\ Pos_{\rho(1)},\ldots,Pos_{\rho(N)}),Len]),$$

where A_0 is the state label of the controller, and $A_1,\ldots,A_N$ are state labels of the trains. So here the state label of the controller as well as the variable Len retains their value, while other variables are permuted by some permutation ρ. For instance, the symmetry definition for Bridge Crossing Problem with 2 trains can be represented as follows:

$$cs([A_0,A_1,A_2],[X_1,X_2,Pos_1,Pos_2,Len]) \models \\ cs([A_0,A_2,A_1],[X_2,X_1,Pos_2,Pos_1,Len]).$$

Dining Philosophers with Timeout. We include this example because its concerns "rotational" symmetry as opposed to "permutation" symmetry. Permutation symmetry is addressable by syntactic means, this kind of symmetry is implemented in a version of Uppaal and in RED.

See Figure 10, where the number N of processes is 3 or more. The boolean F_i indicate if fork i is used. Philosopher i eats by first assigning F_i and $F_{(i+1)\%N}$ to 1. To avoid deadlock, a philosopher would reset F_i to 0 if it cannot set $F_{(i+1)\%N}$ in less than 2 time units. Consider $N=3$, and the assertion

$$cs([P_0,P_1,P_2],[F_0,F_1,F_2,X_0,X_1,X_2]) \models \\ cs([P_0,P_2,P_1],[F_0,F_2,F_1,X_0,X_2,X_1]).$$

The conclusion represents a swap of philosophers 1 and 2. Under permutation symmetry, this assertion holds. But in this case, it does not: suppose that $P_0 = 2$, $P_1 = 0$, $P_2 = 0$,

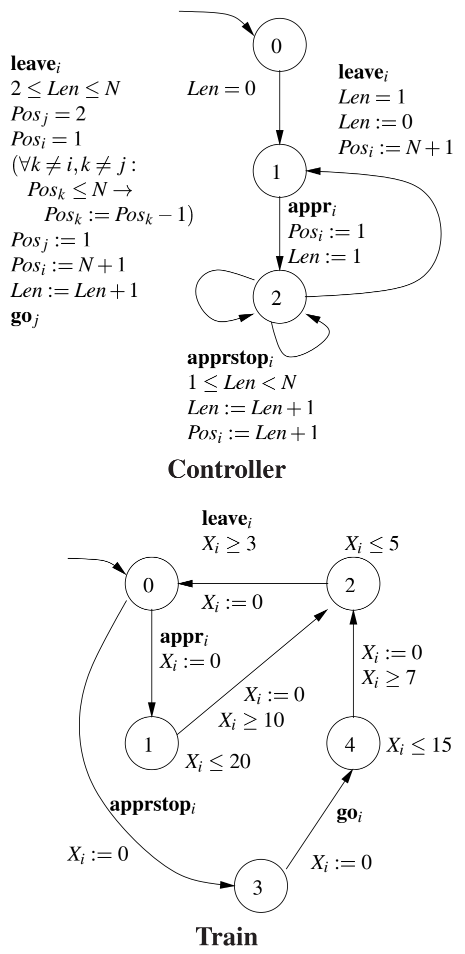


Figure 9. Bridge Crossing Automata

$F_0 = 1$, $F_1 = 0$, $F_2 = 1$. This is an instance of the premise corresponding to philosopher 0 eating and holding both forks. In contrast, the conclusion states that philosopher 0 is eating while only holding one fork!

We now consider "rotational" symmetry, and in fact the following assertion is true:

$$cs([P_0, P_1, P_2], [F_0, F_1, F_2, X_0, X_1, X_2]) \models cs([P_2, P_0, P_1], [F_2, F_0, F_1, X_2, X_0, X_1]).$$

In our experiments, we verify that the number of philosophers in state 2 is never more than $N/2$.

Results. We summarize our results in Table 1. We ran the examples with and without assistance of symmetry, and we considered both time and state space. For the latter, the numbers indicate nodes which are actually stored (the table of assumed assertions), while the numbers in parentheses depict the nodes visited, but not stored because of redundancy. The state space is thus the sum of these two numbers.

Some of our timings are competitive with well known

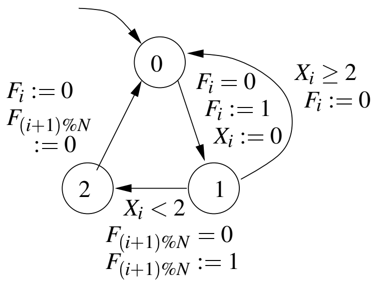


Figure 10. Real-Time Dining Philosophers

systems. In some problems, we are not so competitive. For example, the RED system [18] ran up to 13 processes for Fischer (without using symmetry). Indeed, Hendriks et al. [7], exploiting symmetry along the lines of [11], ran a 100 process version of Fischer! This is certainly out of reach of all the systems we have considered. However, the approaches in these examples are not readily applicable in the general case. Similarly, the XMC/RT and Uppaal systems, though considerably developed, do not have the generality of our relative safety assertions.

The important metric in our experiments is the state space. Our results show a promising level of scalability as a result of using symmetry-like assertions. We believe that there is a rich potential to go beyond these examples and develop new non-behavioral properties than can further reduce the state space.

Acknowledgments

We are grateful to Wang Yi and Giridhar Pemmasani for their valuable comments and help with our experiments.

References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[2] J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. A. Henzinger, editors, *8th CAV*, number 1102 in Lecture Notes in Computer Science, pages 244–256. Springer, 1996.

[3] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms, and tools. http://user.it.uu.se/~yi/ps-files/chapter.ps.

[4] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[5] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *21st RTSS*, pages 175–184. IEEE Computer Society Press, 2000.

	CLP with Coinductive Tabling				XMC/RT		Uppaal
	Time (secs.)		States		Time	States	Time
	Symm.	No Symm.	Symm.	No Symm.	(secs.)		(secs.)
Fischer 4	0.03	1.66	72 (116)	875 (949)	0.82	632	0.04
Fischer 5	0.13	203.47	165 (363)	6872 (9967)	8.91	6330	0.75
Fischer 6	0.42	∞	325 (909)	∞	187.16	75972	71.46
Fischer 7	1.41		591 (1967)		∞	∞	13520.91
Fischer 8	6.17		1016 (3830)				∞
Fischer 9	37.79		1649 (6900)				
Fischer 10	322.76		2536 (14218)				
Fischer 11	3176.60		3759 (18813)				
Bridge 4	0.23	2.63	333 (186)	2805 (18)	1.42	1964	0.02
Bridge 5	1.07	91.9	798 (567)	25875 (78)	13.65	12850	0.10
Bridge 6	4.66	∞	1638 (1428)	∞	140.00	94482	0.76
Bridge 7	21.95		3054 (3150)		∞	∞	8.04
Bridge 8	118.17		5397 (6309)				155.45
Bridge 9	852.27		9288 (11802)				∞
Bridge 10	8674.07		15741 (21072)				
Philosopher 3	0.1	0.16	89 (173)	147 (250)	0.16	422	0.01
Philosopher 4	1.11	2.4	322 (807)	640 (1424)	∞	∞	0.02
Philosopher 5	58.29	188.84	2340 (8397)	5776 (19349)			0.04

Table 1. Experimental Results

[6] 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.

[7] M. Hendriks, G. Behrmann, K. G. Larsen, and F. W. Vaandrager. Adding symmetry reduction to Uppaal. In K. G. Larsen and P. Niebert, editors, *1st FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 46–59. Springer, 2003.

[8] T. A. Henzinger, P.-H. Lo, and H. Wong-Toi. HyTech: A model checker for hybrid systems. In O. Grumberg, editor, *9th CAV*, number 1254 in Lecture Notes in Computer Science, pages 460–463. Springer, 1997.

[9] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[11] C. N. Ip and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2):41–75, 1996.

[12] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, May/July 1994.

[13] J. Jaffar, S. Michaylov, and R. H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[14] S. Mukhopadhyay and A. Podelski. Accurate widenings and boundedness properties of timed systems. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *4th Ershov Memorial Conference*, volume 2244 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2001.

[15] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[16] G. Pemmasani, C. R. Ramakrishnan, and I. V. Ramakrishnan. Efficient model checking of real-time systems using tabled logic programming and constraints. In P. J. Stuckey, editor, *18th ICLP*, volume 2401 of *Lecture Notes in Computer Science*, pages 100–114. Springer, 2002.

[17] K. Sagonas, T. Swift, D. S. Warren, J. Freire, P. Rao, B. Cui, E. Johnson, L. de Castro, S. Dawson, and M. Kifer. *The XSB System Version 2.5 Volume 1: Programmer's Manual*, June 2003.

[18] F. Wang. Efficient verification of timed automata with BDD-like data structures. In L. D. Zuck, P. C. Attie, A. Cortesi, and S. Mukhopadhyay, editors, *VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 189–205. Springer, 2003.

[19] F. Wang. Symbolic parametric safety analysis of linear hybrid systems with BDD-like data-structures. In R. Alur and D. A. Peled, editors, *16th CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 295–307. Springer, 2004.

[20] S. Yovine. Kronos: A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer*, 1(1/2):123–133, October 1997.