Hybrid Automata And SAL
Paritosh K. Pandya & P. Vijay Suman


August, 2007

*This work was supported by General Motors India Science Lab sponsored project "Advanced Research on Formal Analysis of Hybrid Systems".
Abstract

Techniques for modeling behavior of timed and hybrid systems using SAL are investigated. Input models consist of networks of timed (hybrid) automata with location invariants as well as synchronized transitions and urgency. Several techniques for model checking the reachability properties of the resulting models are studied and compared.
Contents

1 Introduction ......................................................... 6

2 Modeling Timed Automata in SAL ............................................. 6
   2.1 Modeling Time .................................................. 7
   2.2 Transitions .................................................... 8
      2.2.1 Timed Transition ........................................ 8
      2.2.2 Discrete Transitions .................................... 10
   2.3 Synchronization and Urgency ..................................... 11
      2.3.1 Train-Gate Controller .................................... 12

3 Symbolic Model Checking ............................................... 13

4 Digitization .......................................................... 16

5 Inductive Model Checking ................................................ 17
   5.1 Train Gate Controller .......................................... 18
   5.2 Hybrid Automata ............................................... 19

6 Discussion .............................................................. 21

A Fischer Protocol - Digitized Model .................................... 23

B Train-Gate - Dense Time Model ........................................ 27

C Pursuit - Hybrid Automata Model .................................... 34
1 Introduction

Hybrid automata provide a mathematical model for digital computer systems that interact with an analog environment in real time. Case studies indicate that the model of hybrid automata is useful for the analysis of embedded software and hardware, including distributed processes with drifting clocks, real-time schedulers, and protocols for the control of manufacturing plants, vehicles, and robots. A special subclass of hybrid automata called timed automata are useful for modeling features of real-time systems and their extensions such as price-timed, stopwatch automata which can model resource scheduling.

In this report, we consider the formal modeling and automatic analysis of timed and hybrid automata using the leading open source tool SAL. SAL is a rich modeling language with support for dynamic types, instantiable modules and contexts. The heart of SAL (Symbolic Analysis Laboratory) is in its capability to specify concurrent systems in a compositional way. SAL supports better type system, better parametric modules and better parametric contexts compared to many existing modeling languages. SAL supports a wide variety of model checking techniques including explicit state-space exploration, symbolic model checking and bounded model checking. It uses the state-of-the-art SMT solver Yices for carrying out bounded model checking and induction proofs.

In the report, we illustrate how to model a network of interacting timed (hybrid) automata in SAL. Behavior of a timed automaton consists of alternating sequence of time elapse transition and discrete transition. We encode this faithfully in the SAL semantics. We consider a rich modeling language where features such as location invariants, synchronous communication as well as urgency (which are also available in the leading timed model checker Uppaal) are supported.

One feature of our modeling is that we allow models where time is dealt with in a parametric fashion. The time can either be discrete or dense, and clocks can be saturated or unsaturated.

Having modeled the timed (hybrid) automata in SAL, we investigate the use of various model checking techniques to analyze reachability and invariance properties of these models. The explored methods are:

- BDD-Based Symbolic Model Checking (SMC) of models with saturated discrete time.
- Inductive model checking using k-induction of models with unsaturated dense time.
- Digitization of Dense-timed automata: This conservatively reduces the problem of checking reachability in dense time automata to checking of reachability in discrete time automata. The SMC technique can be used for the reduced automata.

We provide a comparative experimental evaluation of the above techniques. Several examples of timed and hybrid systems drawn from literature are used for the experimental work.

2 Modeling Timed Automata in SAL

We demonstrate how to model a timed system using SAL. We assume that the input system is described as a network $A_1 \| \ldots \| A_n$ of timed automata. The translated system is an asynchronous parallel composition of the following SAL modules.

- One module called clock which simulates the time elapse transitions. Each such transition non-deterministically picks a value of time step such that location invariants and urgency requirements are met. Values of all clocks are incremented by this step.
A module instance $M_i$ which simulates the local (unsynchronized) transitions of the automaton $A_i$.

A module instance $Sync_{i,j}$ which simulates the synchronized transitions between the automata $A_i$ and $A_j$.

Hence the SAL system is the asynchronous parallel composition

$$\text{system} = \text{clock} || M_i || Sync_{i,j}$$

We illustrate the modeling of Network of Timed Automata in SAL using some examples.

**Fischer Mutual Exclusion Protocol** This timed protocol consists of $n$ processes which must have exclusive access to critical state. The processes execute concurrently in an interleaved fashion. The timed automaton for an individual process under the fischer protocol is shown in Figure 1 (exported from the tool Uppaal-4.0.3). The values of the constants $a, b$ used in the model determine the correctness. Note that locations have invariants which give conditions on clocks under which the process may stay in a location. Also note that the concurrent processes share a variable $id$. In this example, there is no synchronization between the processes.

![Figure 1: Individual Process In Fischer Protocol](image)

In the following we show how the Fischer protocol system is implemented as a SAL model. The code which is given in parts below can be found in Appendix A. The reader is advised to go through the language manual of SAL, while trying to understand the code. But one can easily appreciate the language, given their familiarity with similar tools like SMV.

Each SAL file has a context in it. The context encapsulates the type definitions, the function definitions, the definitions of the modules, the definition of the composite system, and the properties to be verified. The language admits shuffling the order between these, within the context. Note that the contexts in the code given at the above link are parameterized by the number of processes. We quote parts of code in the following. For the convenience of presentation, teletype-font is used for the code.

2.1 Modeling Time

Our models are parameterized by the nature of time. The time can either be discrete or dense, and clocks can be modeled as saturating or unsaturated.

1. Dense Unsaturated Time: The time steps can span arbitrary non-negative real values. Moreover, the clock values can also take arbitrary non-negative values. The function $\text{satinc}(A,B)$ implements adding time values $A$ and $B$. 

7
2. Discrete Saturated Time: The time steps are assumed to take a non-negative integer values. Moreover, there is a maximum constant $M$ at which the clocks in the automata are saturated. It is well known that timed automata with saturated and unsaturated clocks have the same behavior, provided the saturation value $M$ is larger than any constant appearing in the guards and invariants of the timed automata [2].

$$\text{MAXTIME: NATURAL = } M;$$

$$\text{TIME: TYPE = } \{x: \text{NATURAL} \mid x \geq 0 \text{ AND } x \leq \text{MAXTIME}\};$$

$$\text{satinc}(A: \text{TIME}, B: \text{TIME}): \text{TIME} =$$

$$\text{IF } A+B < \text{MAXTIME} \text{ THEN } A+B \text{ ELSE MAXTIME ENDIF;}$$

Advantage of having saturated, discrete time is that it makes the system model finite state.

### 2.2 Transitions

Behavior of a timed automaton consists of alternating sequence of time elapse transition and discrete transition. We encode this faithfully in the SAL model by using a boolean variable \text{timestep} so that there is strict alternation between the timed and discrete transitions of the automaton. One should note that this method in combination with invariants may induce some unintended deadlock steps but it does not affect the reachability/invariance properties.

#### 2.2.1 Timed Transition

The following code is picked from the module \text{clock}. This module updates all the variables that carry the clock values of the automata being simulated. The variables $N$, $a$ and $b$ are the parameters of the context, representing the number of processes and the constants used in the protocol. Each process has a clock $x$ and a location counter, which we call $pc$ (See Figure 1. In our clock model there is an array of clocks and $pc$'s, in each of which the $i$-th element represents the clock and $pc$ of the $i$-th process. We are implementing the system with bounded discrete time. So the discrete step can be of any length $l \in \mathbb{N}$ such that each clock valuation can be incremented without violating the invariants.

The function \text{inv_is_satisfied} checks whether the set of clock valuations satisfy the invariants, given the location counter of each process. The function \text{satinc} which returns the value $\text{max}(A+B, \text{MAXTIME})$, where all the values are of datatype \text{TIME}, is used to increment the clocks. Note that, one can easily change the datatype \text{TIME} and this function to model the protocol in a different time domain. The function \text{incr_clocks} actually computes and returns the array of clocks in which each clock is incremented by the time step. For hybrid automata this function is considerably more complicated.

```sal
dfischer\{N: \{x: \text{NATURAL} \mid x \geq 2\}, a: \text{NATURAL}, b: \text{NATURAL}\} : \text{CONTEXT} =
BEGIN
% Some typedefinitions and global variables
PROC_INDEX: \text{TYPE} = [1..N];
TEMP_OWNER: \text{TYPE} = [0..N];
```
MAXTIME: NATURAL = max(a,b) + 1;
TIME: TYPE = { x: NATURAL | x >= 0 AND x <= MAXTIME }; 
% TIME: TYPE = { x: REAL | x >= 0};
PC: TYPE = { sleep, req, wait, critical }; 
CLOCK_ARRAY: TYPE = ARRAY PROC_INDEX OF TIME;
PC_ARRAY: TYPE = ARRAY PROC_INDEX OF PC;

% ----------------------

satinc(A:TIME,B:TIME):TIME = 
  IF A+B < MAXTIME THEN A+B ELSE MAXTIME ENDIF;

incr_clocks(x_array: CLOCK_ARRAY, idx: PROC_INDEX, step: TIME): CLOCK_ARRAY = 
  IF idx = N THEN x_array WITH [N] := satinc(x_array[N],step) 
  ELSE incr_clocks((x_array WITH [idx] := satinc(x_array[idx],step)), 
                  idx+1, step) 
  ENDIF;

inv_is_satisfied(pc_array: PC_ARRAY, x_array: CLOCK_ARRAY, 
                  idx: PROC_INDEX): BOOLEAN = 
  IF idx = N THEN
    ((pc_array[N] = sleep OR 
      pc_array[N] = wait OR 
      pc_array[N] = critical) OR 
    (pc_array[N] = req AND x_array[N] <= a) 
  )
  ELSE 
    (((pc_array[idx] = sleep OR 
      pc_array[idx] = wait OR 
      pc_array[idx] = critical) OR 
    (pc_array[idx] = req AND x_array[idx] <= a) 
  ) AND 
    inv_is_satisfied(pc_array, x_array, idx + 1) 
  )
  ENDIF;

The clock module has just one transition, which checks whether the invariant is met after the timed transition is taken with the non-deterministically selected step value and it increments the clocks by that step. The boolean variable timestep, is used to strictly alternate between discrete and timed transitions. The variable temp_owner has the value of the (global) variable id of the automaton model.

% The module defining the timed transition of the system
clock: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL temp_owner: TEMP_OWNER
GLOBAL timestep: BOOLEAN
LOCAL step: TIME
INITIALIZATION

\[ \text{temp\_owner} = 0; \text{timestep} = \text{TRUE}; \]
\[ (\text{FORALL} \ (i: \text{PROC\_INDEX}): \text{x\_array}[i] = 0) \]

TRANSITION

% Must determine a nondeterministic step value that satisfies the invariant
% and advance all clocks
\[ \text{timestep AND} \]
\[ \text{inv\_is\_satisfied(pc\_array,x\_array',1)} \rightarrow \]
\[ \text{step' IN \{ x:TIME | true \}}; \]
\[ \text{x\_array'} = \text{incr\_clocks(x\_array,1, step')}; \]
\[ \text{timestep' = FALSE}; \]

END;

2.2.2 Discrete Transitions

Let \( T = T_1, \ldots, T_l \) be the set of local (unsynchronized) transitions of a timed automaton. Each transition \( T_i \) is encoded in SAL and is denoted by \( \text{encode}(T_i) \). The guard of \( \text{encode}(T_i) \) is the conjunction of conditions which enable the transition (which always includes \( \text{NOT(timestep)} \)). The assignment dictates the actual change in the state because of the transition being taken. Note that assignment must also check whether the invariant is satisfied in the resultant state. The transition relation of the module \( M \) simulating the actual (independent) discrete moves of the automaton \( A \) will look as follows:

\[ \text{TRANSITION} \ [\text{encode}(T_1) \ [ \ldots [ \text{encode}(T_l)]] \]

The following code extracted from \texttt{dfischer.sal} represents the transition relation simulating the independent transitions of an individual Fischer process.

TRANSITION

\[
\begin{align*}
\text{waking\_up:} & \quad \text{NOT(timestep) AND} \\
& \quad \text{pc\_array}[i] = \text{sleep AND} \\
& \quad \text{temp\_owner} = 0 \rightarrow \\
& \quad \text{pc\_array'}[i] = \text{req}; \\
& \quad \text{x\_array'}[i] = 0; \\
& \quad \text{timestep'} = \text{TRUE}; \\
\end{align*}
\]

\[
\begin{align*}
\text{leave\_critical:} & \quad \text{NOT(timestep) AND} \\
& \quad \text{pc\_array}[i] = \text{critical} \rightarrow \\
& \quad \text{pc\_array'}[i] = \text{sleep}; \\
& \quad \text{temp\_owner'} = 0; \\
& \quad \text{timestep'} = \text{TRUE}; \\
\end{align*}
\]
The encoding of the transition relation when there is synchronization and urgency is described in the following sections. Here we consider the case where only two processes synchronize at a time as in CSP or Uppaal.

### 2.3 Synchronization and Urgency

In networks of timed automata, synchronization is specified using transitions with the same labels.

**2-Doors Synchronization** Consider the model of Two Doors Controller shown in Figure 2 (exported from the tool Uppaal-4.0.3). The idea is that the two doors are never open at the same time. The door and its corresponding user synchronize on the event `pushed` and the two doors synchronize on the event `closed`.

![Figure 2: The automata for the two doors example](image)

Also note that all the transitions labeled with events `pushed` and `closed` are *urgent*. Hence the length of the timestep is forced to be 0, when one of these transitions is enabled. When such transitions are part of the system, the timed transition takes a step of length 0. This is illustrated in the code below (see file `d2doors.sal`).

The synchronized transitions between component timed automata $A_i$ and $A_j$ are modeled using the transition relation of a module $\text{Sync}_{i,j}$. A synchronized transition $\text{encode}(T_1, T_2)$ is enabled iff the corresponding transitions $T_1$ and $T_2$ with same label in the component automata $A_i$ and $A_j$ are simultaneously enabled. i.e., the transition needs to be taken simultaneously by all the component automata.

```sal
% The module for synchronized transitions between user1 and door1
userdoor1: MODULE =
BEGIN
BEGIN

% ONLY SYNCHRONIZED TRANSITIONS BETWEEN DOOR1 TO DOOR2
```
TRANSITION
[

pushed_busy_to_idle_and_idle_to_wait:
  doorpc1 = idle AND userpc1 = busy AND
  NOT(timestep) -->
  doorpc1' = wait;
  userpc1' = idle;
  activated1' = TRUE;
  timestep' = TRUE;
]
END;

Also note that all the transitions labeled with events are urgent. Hence the length of the timestep
is forced to be 0, when one of these transitions is enabled. This is modeled in SAL as follows. The
time-elapse transition is modified such that the value of the “step” is forced to zero whenever an
urgent transition is enabled.

DEFINITION
urgent_enabled =
  (doorpc1=idle AND userpc1=busy) OR
  (doorpc2=idle AND userpc2=busy) OR
  (doorpc2=wait AND (doorpc1=idle OR doorpc1=wait OR doorpc1=closed)) OR
  (doorpc1=wait AND (doorpc2=idle OR doorpc2=wait OR doorpc2=closed));

TRANSITION
[ timelapse: timestep AND doorinv_is_satisfied(doorpc1,x1') AND
  doorinv_is_satisfied(doorpc2,x2') AND
  urgent_enabled => step'=0) -->
  step' IN { x:TIME | true };
  x1'=inctime(x1,step');
  x2'=inctime(x2,step');
  w1'=inctime(w1,step');
  w2'=inctime(w2,step');
  timestep' = FALSE;
]

The composite model is defined as composition of instances of modules. The composition and module
instantiation of SAL is pretty rich. One can instantiate the modules with variable renaming to get
copies of it with different names for the variables. Renaming of variables can be used in order to
share variables among module instances. Refer to the code for the model of two doors given in the
file d2doors.sal at the afore-mentioned web address, for an example of usage of renaming.

2.3.1 Train-Gate Controller

The Train-Gate model is parameterized by the number of trains \( N \). \( N \) trains try to cross a com-
mon track. The signalling system coordinates their activity by putting approaching trains into a
FIFO queue. At a time only one train can cross. When a train finishes crossing, the controller dis-
patches the longest waiting train for crossing. We can implement this using a queue data type. The
SAL-code for simulating the Train-Gate controller is given in Appendix B. The module instance
train[i] simulates the independent transitions of the automaton Train. And each of appr[i], stop[i], leave[i] and go[i] are the modules simulating the synchronized transitions between the instance i of the Train and the automaton Gate. Note that there are no independent transitions in the automaton Gate. The following is the asynchronous parallel composition in the case of this model.

\[
\text{trainsgate: MODULE = } \left( \left[ \right. \right. (i: \text{TRAIN_INDEX}): (\text{train[i] [] appr[i] [] stop[i] [] leave[i] [] go[i]}));
\]

\[
\text{system: MODULE = clock [] trainsgate;}
\]

3 Symbolic Model Checking

When timed automata are modeled in SAL using discrete, saturated time the clocks take values from a subrange of integers. As a result the system can be considered as a finite state system and the familiar bdd-based symbolic model checking techniques can be used to verify reachability and invariance properties. SAL Engine provides the sal-smc command for performing such model checking.

Consider the Fischer mutual exclusion model described earlier. The mutual exclusion property that no two processes are in critical state simultaneously can be specified in SAL as

\[
\text{mutex: THEOREM}
\]

\[
\text{system |- G (FORALL (i, j: \text{TRAIN_INDEX})):}
\]

\[
i/=j \Rightarrow \text{NOT(pc_array[i] = cross AND pc_array[j] = cross});
\]

This invariant requirement can be checked by invoking the sal engine command
In this section we quote the figures for symbolically model checking these properties. All the following experiments were run on a 2.93 GHz Intel Pentium 4 processor with 1 GB of main memory, and running Fedora Core release 2 (Tettnang).

The time it takes to model check the mutual exclusion property for fischer protocol depends on the constants $a$ and $b$. We quote four tables (Tables 1 through 4) of performance for the symbolic model checking fischer protocol for mutual exclusion for different values of $a$ and $b$.

Table 1: Mutual Exclusion In Fischer Protocol With $N$ Processes, $a=05$ and $b=06$ - (SMC)

<table>
<thead>
<tr>
<th>N</th>
<th>Composition</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>asynchronous</td>
<td>0.10</td>
<td>1748</td>
<td>626</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.09</td>
<td>6325</td>
<td>5272</td>
</tr>
<tr>
<td>3</td>
<td>asynchronous</td>
<td>0.19</td>
<td>22014</td>
<td>6229</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.15</td>
<td>20440</td>
<td>73836</td>
</tr>
<tr>
<td>4</td>
<td>asynchronous</td>
<td>0.35</td>
<td>49232</td>
<td>65464</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.47</td>
<td>62220</td>
<td>967472</td>
</tr>
<tr>
<td>5</td>
<td>asynchronous</td>
<td>2.09</td>
<td>135794</td>
<td>686195</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>1.84</td>
<td>177668</td>
<td>11770500</td>
</tr>
<tr>
<td>6</td>
<td>asynchronous</td>
<td>20.16</td>
<td>858023</td>
<td>7068366</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>8.61</td>
<td>588737</td>
<td>134327496</td>
</tr>
<tr>
<td>7</td>
<td>asynchronous</td>
<td>101.51</td>
<td>4199722</td>
<td>71495765</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>40.64</td>
<td>1229262</td>
<td>1457831508</td>
</tr>
<tr>
<td>8</td>
<td>asynchronous</td>
<td>451.54</td>
<td>10094966</td>
<td>711783444</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>420.43</td>
<td>4279742</td>
<td>15216921952</td>
</tr>
</tbody>
</table>

Table 2: Mutual Exclusion In Fischer Protocol With $N$ Processes, $a=09$ and $b=10$ - (SMC)

<table>
<thead>
<tr>
<th>N</th>
<th>Composition</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>asynchronous</td>
<td>0.11</td>
<td>2575</td>
<td>1266</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.09</td>
<td>5690</td>
<td>11208</td>
</tr>
<tr>
<td>3</td>
<td>asynchronous</td>
<td>0.18</td>
<td>17474</td>
<td>18927</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.26</td>
<td>47377</td>
<td>248400</td>
</tr>
<tr>
<td>4</td>
<td>asynchronous</td>
<td>1.21</td>
<td>67785</td>
<td>297828</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>1.77</td>
<td>18465</td>
<td>5047728</td>
</tr>
<tr>
<td>5</td>
<td>asynchronous</td>
<td>8.34</td>
<td>430970</td>
<td>4624765</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>15.42</td>
<td>657955</td>
<td>93416400</td>
</tr>
<tr>
<td>6</td>
<td>asynchronous</td>
<td>74.53</td>
<td>2694075</td>
<td>70246818</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>61.50</td>
<td>2895884</td>
<td>1609723080</td>
</tr>
<tr>
<td>7</td>
<td>asynchronous</td>
<td>760.30</td>
<td>22033350</td>
<td>1045685995</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>472.71</td>
<td>9308409</td>
<td>26318751984</td>
</tr>
</tbody>
</table>

The two doors model is not parametric. We model-checked some of the properties of the two doors model. These are listed in Table 5. A different model with an extra variable $y$ was needed to verify the property wait_to_closed.
Table 3: Mutual Exclusion In Fischer Protocol With $N$ Processes, $a=13$ and $b=14$ - (SMC)

<table>
<thead>
<tr>
<th>$N$</th>
<th>Composition</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>asynchronous synchronous</td>
<td>0.11</td>
<td>2925</td>
<td>2130</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.09</td>
<td>8784</td>
<td>13989</td>
</tr>
<tr>
<td>3</td>
<td>asynchronous synchronous</td>
<td>0.19</td>
<td>19614</td>
<td>42713</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.31</td>
<td>51147</td>
<td>590484</td>
</tr>
<tr>
<td>4</td>
<td>asynchronous synchronous</td>
<td>1.78</td>
<td>96071</td>
<td>890288</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>3.23</td>
<td>329927</td>
<td>1616576</td>
</tr>
<tr>
<td>5</td>
<td>asynchronous synchronous</td>
<td>15.57</td>
<td>746345</td>
<td>18189895</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>37.83</td>
<td>1525979</td>
<td>399699740</td>
</tr>
<tr>
<td>6</td>
<td>asynchronous synchronous</td>
<td>198.50</td>
<td>5670735</td>
<td>363041254</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>299.24</td>
<td>7503062</td>
<td>9184664136</td>
</tr>
</tbody>
</table>

Table 4: Mutual Exclusion In Fischer Protocol With $N$ Processes, $a=17$ and $b=18$ - (SMC)

<table>
<thead>
<tr>
<th>$N$</th>
<th>Composition</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>asynchronous synchronous</td>
<td>0.13</td>
<td>4900</td>
<td>3218</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.09</td>
<td>7576</td>
<td>29800</td>
</tr>
<tr>
<td>3</td>
<td>asynchronous synchronous</td>
<td>0.96</td>
<td>49001</td>
<td>81043</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>0.49</td>
<td>79210</td>
<td>1156536</td>
</tr>
<tr>
<td>4</td>
<td>asynchronous synchronous</td>
<td>4.91</td>
<td>165647</td>
<td>2097052</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>8.83</td>
<td>586333</td>
<td>39738416</td>
</tr>
<tr>
<td>5</td>
<td>asynchronous synchronous</td>
<td>74.39</td>
<td>1309739</td>
<td>52986705</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>64.54</td>
<td>3741554</td>
<td>1227752040</td>
</tr>
<tr>
<td>6</td>
<td>asynchronous synchronous</td>
<td>675.37</td>
<td>10663473</td>
<td>1307406234</td>
</tr>
<tr>
<td></td>
<td>synchronous</td>
<td>755.84</td>
<td>21268490</td>
<td>35234418504</td>
</tr>
</tbody>
</table>
Table 5: Two Doors Mutual Exclusion Protocol - (SMC)

<table>
<thead>
<tr>
<th>Property</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual Exclusion</td>
<td>0.23</td>
<td>26399</td>
<td>4765</td>
</tr>
<tr>
<td>Bounded Time Condition</td>
<td>0.23</td>
<td>26393</td>
<td>4765</td>
</tr>
<tr>
<td>Door1 Can Open</td>
<td>0.62</td>
<td>45676</td>
<td>7404</td>
</tr>
<tr>
<td>Wait To Closed</td>
<td>4.62</td>
<td>126883</td>
<td>440481</td>
</tr>
</tbody>
</table>

The most involved example considered was the train-gate. The performance of verification in train-gate protocol is parameterized by the number of the trains in the model. Note that the corresponding context was parameterized by this variable \( N \). The figures of performance on the same machine are listed in Table 6.

Table 6: Mutual Exclusion In Train-Gate Protocol With \( N \) Trains - (SMC)

<table>
<thead>
<tr>
<th>( N )</th>
<th>Time(S)</th>
<th>BDD</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.16</td>
<td>10573</td>
<td>5893</td>
</tr>
<tr>
<td>3</td>
<td>3.63</td>
<td>193915</td>
<td>310750</td>
</tr>
<tr>
<td>4</td>
<td>145.61</td>
<td>5594103</td>
<td>1564587</td>
</tr>
</tbody>
</table>

4 Digitization

Symbolic bdd-based verification of invariance and LTL properties is an effective technique for discrete, saturated time models. In order to extend the applicability of these techniques to dense-time models we can use digitization.

In this section we see how to approximately reduce the model checking of dense-time properties to model checking of discrete-time properties. We illustrate the technique using the Fischer protocol example. The digitized code for this example is given in Appendix A.

Theorem 4.1. ([4]) Given a timed automaton \( A \) (where the constants in the guards are integers), and a location reachability property \( \varphi \) (in terms of location), the question whether \( L(A) \models \varphi \) is equivalent to \( L(A) \models \varphi \) provided \( A \) is closed. By closed model \( A \), we mean that the set of clock constraints which occur in guards and invariants are conjunctions of formulae of the form \( x \leq c \) or \( x \geq c \).

Consider the fischer protocol as modeled in Uppaal (see Figure 1). In the actual model, the invariant on the location \texttt{req} and the guard on the transition from \texttt{req} to \texttt{wait} are of the form \( x=a \), which are closed constraints. The guard on the transition from \texttt{wait} to \texttt{cs} is \( x>b \), which is an open constraint. In order to make the automaton model closed, we replace the later guard \( x>b \) by a guard of the type \( x\geq k \) with suitable choice of \( k \).

1. If \( k \) is chosen to be \( b \) we get a guard \( x\geq b \) which is weaker than the original guard and we have over-approximated the automaton which now has more behaviours. Proving validity of property against the over-approximated model also proves its validity in the original case.
2. If \( k \) is chosen to be \( b + 1 \) we get a guard \( x > b + 1 \) which is stronger than the original guard, and we have under-approximated the automaton which now has fewer behaviours. Finding counter-example for the property against the under-approximated model also gives a counter-example in the original model.

3. In both above case, the resulting model is closed and hence by digitization theorem, it can be analysed in discrete-time.

While the digitization technique is approximate we found that it works quite effectively in many examples of interest. Unfortunately, the digitization method does not seem to extend very easily to either hybrid automata or even to its special cases such as stopwatch automata [6].

5 Inductive Model Checking

Now we consider, model checking the invariance properties of timed automata with unsaturated real-time using the principle of \( k \)-induction. We first describe this principle. The fact that \( P \) is invariably true is denoted by the LTL formula \( GP \).

Let \( X \) be the list of system variables. In inductive model checking, the invariant \( P(X) \) to be established is checked as follows. Let \( T(X, X') \) be the transition relation of the system and let \( I(X) \) give the set of initial states. Then, proving \( GP \) reduces to checking the validity of the “math-sat” formulae \( P(X) \land T(X, X') \Rightarrow P(X') \) and \( I(X) \Rightarrow P(X) \). This is generalized to \( k \)-induction with induction depth \( k \). Under this principle, if the following formulae can be shown valid then we can conclude \( GP \).

\[
I(X_0) \land T(X_0, X_1) \land \ldots \land T(X_{k-1}, X_k) \Rightarrow P(X_0) \land \ldots \land P(X_k) \\
P(X_0) \land T(X_0, X_1) \land P(X_1) \land T(X_1, X_2) \land \ldots \land P(X_{k-1}) \land T(X_{k-1}, X_k) \Rightarrow P(X_k)
\]

Since these two formulae are quantifier-free formulae, the SMT solver incorporated in SAL can often prove these steps automatically. The \( k \)-induction based checking of a property \( GP \) can be carried out in SAL using the command `sal-inf-bmc -d k`.

We apply the \( k \)-induction based model checking to several examples discussed earlier.

- **Fischer Protocol**: Table 7 gives the time taken to establish mutual exclusion of \( N \) process Fischer protocol using \( k \)-induction. An instance \( N(A,B) \) denotes an \( N \) process Fischer protocol with constants \( A,B \) used in the code.

<table>
<thead>
<tr>
<th>( N(a,b) )</th>
<th>Asynchronous</th>
<th>Synchronous</th>
</tr>
</thead>
<tbody>
<tr>
<td>3(5,6)</td>
<td>0.63</td>
<td>0.79</td>
</tr>
<tr>
<td>4(5,6)</td>
<td>1.56</td>
<td>1.86</td>
</tr>
<tr>
<td>5(5,6)</td>
<td>5.62</td>
<td>5.20</td>
</tr>
<tr>
<td>6(5,6)</td>
<td>19.52</td>
<td>18.20</td>
</tr>
<tr>
<td>6(50,51)</td>
<td>21.47</td>
<td>18.25</td>
</tr>
<tr>
<td>6(500,600)</td>
<td>19.52</td>
<td>19.63</td>
</tr>
<tr>
<td>7(5,6)</td>
<td>96.35</td>
<td>132.21</td>
</tr>
</tbody>
</table>
• **Two Doors:** Table 8 gives the time taken to verify the 2-doors problem.

Table 8: Two Doors Protocol

<table>
<thead>
<tr>
<th>Property</th>
<th>Time(S)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutual Exclusion</td>
<td>0.54</td>
<td>9</td>
</tr>
</tbody>
</table>

5.1 **Train Gate Controller**

The model of Train Gate controller was given earlier. The model is parameterized by the number of trains N. As stated before N trains try to cross a common track. Signalling system coordinates their activity by putting approaching trains into a FIFO queue. At a time only one train can cross. When a train finishes crossing, the controller dispatches the longest waiting train for crossing. We can implement this using a queue data type. The model for dense time is given in Appendix B.

The “mutex” property that at most one train can be crossing at a time is specified by the following invariant.

\[
\text{mutex: THEOREM} \\
\text{system} |- \ G(\text{FORALL (i, j: TRAIN_INDEX): i=/=j} => \ NOT(pc\_array[i] = \text{cross} AND pc\_array[j] = \text{cross}));
\]

This can be verified using k–induction by using the following command. The Table 9 gives the time taken and the induction depth necessary for proving the mutex property.

```
sal-inf-bmc -v 10 -i -d 30 --assertion='ntraingate=3 mutex'
```

Table 9: Inductive Verification of Train Gate Protocol

<table>
<thead>
<tr>
<th>N</th>
<th>Time(S)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.58</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>331.44</td>
<td>30</td>
</tr>
</tbody>
</table>

In a variant of the train gate controller model, in place of queue data structure, each train is assigned a token denoting its position in the queue. If the train is not in queue it is assigned token 0. The file `ntraingate.sal` implements this model. The Table 10 gives the time taken and induction depth needed to verify modified train gate model. Unfortunately, the naive inductive model checking does not perform well in this example.

We can define some straightforward invariance properties of the queue encoding used. This is given by the THEOREM `queue1` below. It states that exactly the trains which are in safe state have token 0 (i.e. they are not in queue). Secondly, the number of trains having non-zero tokens equals the
value of the variable queue.count.

queue1: THEOREM
system |- G((FORALL (i:TRAIN_INDEX):
    pc_array[i] = safe <=> queue.queue_array[i] = 0) AND
    (queue_count(queue,1)=queue.count));

Table 11 gives the time taken and the depth necessary to establish the queue invariant using \( k \)-induction.
Moreover, we can now try to establish mutual exclusion assuming this queue invariant. This can be done by giving the command

```
sal-inf-bmc -v 3 -i -d 19 -l queue1 --assertion='ntraingate3mutex'
```

The time taken and induction depth required to prove mutex are given in Table 12. It can be seen that by assuming the obvious queue invariant the time for checking mutex is drastically reduced.

## 5.2 Hybrid Automata

We can also apply \( k \)-induction based model checking to several hybrid automata based models. It must be noted that such models cannot be easily digitized and hence the earlier SMC based approach cannot be used.

<table>
<thead>
<tr>
<th>N</th>
<th>Time(S)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.27</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>769 (failed)</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 10: Inductive Verification of Modified Train Gate Protocol

<table>
<thead>
<tr>
<th>N</th>
<th>Time(S)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4.31</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>224.04</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 11: Proving Queue Invariant in Modified Train Gate Protocol

<table>
<thead>
<tr>
<th>N</th>
<th>Time(S)</th>
<th>Depth</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2.8</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>73.47</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 12: Mutual Exclusion in Modified Train Gate Protocol Assuming Queue Invariant
**Pursuit Game**  The primary difference in modeling hybrid automata from that of modeling timed automaton is how the timed transition effects the variables of the model and the way the kind of guards and updates used over the discrete transitions. Also, the termination of the verification of an assertion on a given hybrid automaton model is not guaranteed. We elucidate the timed transition for hybrid automata with the help of the automaton modeling the pursuit game as in [1] which was described as . . .

There is a pursuer in a golf cart chasing an evader on a circular track 40 meters long. The cart can travel up to 6 meters per second in the clockwise direction, but only up to 1/2 meters per second going counterclockwise, since it must use its reverse gear to travel in this direction. The evader is on a bicycle, and travels at 5 meters per second in either direction. However, it makes a decision whether to change its direction only at fixed points in time, separated by exactly two seconds. The goal of the evader is to avoid the pursuer. The evader has the added advantage that there is a rescue helicopter at, a fixed position on the track.

The safety condition says that the evader reaches the helicopter without crossing the pursuer. Our model is slightly different from the one shown in the figure in the original paper. The rate of the variable \( e \) in the location *counter* (corresponding to the speed of the evader in the counter-clockwise direction) is \(-5\), instead of 5. Nevertheless the model checking procedure is known to terminate for assertions specifying reachability properties.

For example, the timed transition corresponding to this particular automaton model is given below.

```plaintext
TRANSITION
[
  continuous_step:
    timestep = TRUE AND
    step' >= 0 AND
    inv_cc(e',t',p') AND
    2*new_p' >= 2*p - step' AND new_p' <= p + 6*step' -->
    t' = IF loc = rescued THEN t ELSE t + step' ENDIF;
    e' = IF loc = rescued THEN e ELSIF loc = clockwise THEN e + 5*step'
    ELSE e - 5*step'
    ENDIF;
    p' = IF loc = rescued THEN p ELSE new_p' ENDIF;
    timestep' = FALSE;
]
```

Our model satisfies the condition in more cases than the original. E.g. consider the case where the initialization condition is as follows. The model in [1] will not satisfy the safety condition, whereas our model shows that the evader can still win.

```plaintext
INITIALIZATION
  loc = clockwise;
  timestep = TRUE;
  t = 2;
```
The SAL model of hybrid automata corresponding to the pursuit game is given in Appendix C.

### 6 Discussion

We have proposed two approaches to verification of invariance properties of real-time systems. In the first approach, digitization is used to conservatively reduce the dense-time reachability problem to discrete-time reachability problem. Moreover, the clock values can be bounded to maximum constant occurring in the program and the property. Hence, we can perform model checking of the reduced model by symbolic bdd-based model checking. In the other approach, we directly verify the dense-time system using $k$-induction. This approach also can be used with hybrid automata in general. We have profiled the performance of the two approaches using a few problems. We observe the following trends.

- Model checking by $k$-induction is not very sensitive to values of the constants occurring in the model/formula. Hence this approach can be used when constants are large. As against this, the SMC approach is quite sensitive to the constants occurring in the automata and formulae and the approach seems applicable only to instances with small constants.

- Improving the SMC performance by guided search using reachability expressions [7] is an interesting path to pursue.

- Symbolic Model Checking outperforms $k$-induction based checking in case of train gate. However, with some obvious invariants, $k$-induction can be done more effectively. The methods to synthesize such simple invariants need to be investigated.

- The verification techniques need to be extended to assertions other than just invariance properties. SMC can verify the full class of LTL properties of discrete timed automata.

- Digitization can be used to approximately verify the LTL properties of timed automata. Unfortunately this technique does not seem to extend to either hybrid automata or even its special cases such as stopwatch automata [6]. It is a very interesting question to find techniques for conservatively approximating dense-time automata by discrete-time automata.

- As implemented in SAL, the $k$-induction can only verify invariance properties. We are investigating the use of a logic QDDC-LOC which can express safety and bounded liveness properties [5]. Using monitor synthesis for QDDC-LOC formulae, the verification of QDDC-LOC properties of dense-time models can be reduced to invariance checking.

**Related Work** There are many existing model checkers for timed and hybrid systems. Notable amongst them are Uppaal, Kronos and Hytech. All of these are based on symbolic reachability search using Zones (polyhedra, Difference Bound Matrices etc.) The techniques such as $k$-induction supported by SAL are quite different and we have investigated the applicability of these techniques for model checking timed and hybrid systems.

Before us, Duertt and Sorea have also investigated modeling and model checking of timed automata in SAL [3]. Unlike their method, our encoding results into an asynchronous parallel composition of modules. This has the advantage that techniques for efficient state-space exploration such as guided symbolic search can be used. Our future work will explore such optimizations.
References


APPENDIX

The following appendices give the complete SAL code for the example models considered in the report. This code and the code for additional models can be found at the following webpage:

http://www.tcs.tifr.res.in/~vsuman/GM-Sponsored-Project/sal-hybrid/

A Fischer Protocol - Digitized Model

dfischer\{N: \{x: \text{NATURAL} \mid x \geq 2\}, a: \text{NATURAL}, b: \text{NATURAL} \}: \text{CONTEXT} =

BEGIN

% Some typedefinitions and global variables
PROC_INDEX: \text{TYPE} = \{1..N\};
TEMP_OWNER: \text{TYPE} = \{0..N\};
MAXTIME: \text{NATURAL} = \text{max}(a,b) + 1;
TIME: \text{TYPE} = \{x: \text{NATURAL} \mid x \geq 0 \text{ AND } x \leq \text{MAXTIME}\};
PC: \text{TYPE} = \{\text{sleep, req, wait, critical}\};
CLOCK_ARRAY: \text{TYPE} = \text{ARRAY\ PROC_INDEX\ OF\ TIME};
PC_ARRAY: \text{TYPE} = \text{ARRAY\ PROC_INDEX\ OF\ PC};

% ----------------------

satinc(A:TIME,B:TIME):TIME =
  \text{IF}\ A+B < \text{MAXTIME}\ \text{THEN}\ A+B\ \text{ELSE}\ \text{MAXTIME}\ \text{ENDIF};

incr_clocks(x_array: CLOCK_ARRAY, idx: PROC_INDEX, step: TIME): CLOCK_ARRAY =
  \text{IF}\ idx = N\ \text{THEN}\ x_array\ \text{WITH}\ [N] := \text{satinc}(x_array[N],step)
  \quad \text{ELSE}\ \text{incr_clocks}(
    (x_array\ \text{WITH}\ [idx] := \text{satinc}(x_array[idx],step)),
    idx+1,\ step)
  \text{ENDIF};

inv_is_satisfied(pc_array: PC_ARRAY, x_array: CLOCK_ARRAY, idx: PROC_INDEX): BOOLEAN =
  \text{IF}\ idx = N\ \text{THEN}
    ((pc_array[N] = \text{sleep OR}
    \quad pc_array[N] = \text{wait OR}
    \quad pc_array[N] = \text{critical})\ \text{OR}
    \quad (pc_array[N] = \text{req AND x_array[N] \leq a})
    )
  \text{ELSE}
    (\text{inv_is_satisfied}(pc_array[N-1], x_array[N-1], N-1))
  \text{ENDIF};

END
pc_array[idx] = wait OR
pc_array[idx] = critical) OR
(pc_array[idx] = req AND x_array[idx] <= a) AND
inv_is_satisfied(pc_array, x_array, idx + 1)
ENDIF;

% ----------------------
% The module defining the timed transition of the system

clock: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL temp_owner: TEMP_OWNER
GLOBAL timestep: BOOLEAN
LOCAL step: TIME

INITIALIZATION
  temp_owner = 0; timestep = TRUE;
  (FORALL (i: PROC_INDEX): x_array[i] = 0)

TRANSITION
% Must determine a nondeterministic step value that satisfies the invariant
% and advance all clocks

  [ timestep AND inv_is_satisfied(pc_array, x_array', 1) -->
    step' IN { x:TIME | true }; x_array' = incr_clocks(x_array, 1, step');
    timestep' = FALSE;
  ]
END;

% ----------------------
% The module defining an individual process

process[i: PROC_INDEX]: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL temp_owner: TEMP_OWNER
GLOBAL timestep: BOOLEAN

INITIALIZATION
  pc_array[i] = sleep;
TRANSITION
[

waking_up:
NOT(timestep) AND
pc_array[i] = sleep AND
temp_owner = 0 -->
 pc_array'[i] = req;
 x_array'[i] = 0;
 timestep' = TRUE;
]

get_temp_ownership:
NOT(timestep) AND
pc_array[i] = req AND
x_array[i] <= a -->
 pc_array'[i] = wait;
 x_array'[i] = 0;
 temp_owner' = i;
 timestep' = TRUE;
[]

loose_temp_ownership:
NOT(timestep) AND
pc_array[i] = wait AND
temp_owner = 0 AND
x_array'[i] <= a -->
 pc_array'[i] = req;
 x_array'[i] = 0;
 timestep' = TRUE;
]

enter_critical:
NOT(timestep) AND
pc_array[i] = wait AND
temp_owner = i AND
x_array[i] >= b -->
 pc_array'[i] = critical;
 timestep' = TRUE;
[]

leave_critical:
NOT(timestep) AND
pc_array[i] = critical -->
 pc_array'[i] = sleep;
 temp_owner' = 0;
 timestep' = TRUE;
]

END;

% ----------------------
% Definitions of the system as an asynchronous compostion of
% processes, and clocks

processes: MODULE =
   [[] (i: PROC_INDEX): process[i]];

system: MODULE = clock [] processes;

% ----------------------

% Mutual Exclusion property to be checked

mutex: THEOREM
   system |- G(FORALL (i, j: PROC_INDEX):
      (i/=j => NOT(pc_array[i] = critical AND pc_array[j] = critical)));

% Process 'i' enters wait state whenever it requests

p1req_wait: THEOREM
   system |- G(pc_array[1] = req => F(pc_array[1] = wait));

END
Train-Gate - Dense Time Model

BEGIN

% Some typedefinitions and global variables

TRAIN_INDEX: TYPE = [1..N];
X_INDEX: TYPE = [0..N];
TIME: TYPE = { x: REAL | x >= 0 };
appr_x_max: TIME = 20;
start_x_max: TIME = 15;
cross_x_max: TIME = 5;

PC: TYPE = { safe, appr, stop, start, cross };
CLOCK_ARRAY: TYPE = ARRAY TRAIN_INDEX OF TIME;
PC_ARRAY: TYPE = ARRAY TRAIN_INDEX OF PC;
GATE_PC: TYPE = { free, occ, more };
QUEUE: TYPE = [# train_array: ARRAY TRAIN_INDEX OF X_INDEX,
  front: TRAIN_INDEX,
  count: X_INDEX #];

% ----------------------
satinc(A:TIME, B:TIME): TIME = A+B;
incr_clocks(x_array: CLOCK_ARRAY, idx: TRAIN_INDEX, step: TIME): CLOCK_ARRAY =
  IF idx = N THEN x_array WITH [N] := satinc(x_array[N], step)
  ELSE incr_clocks(
      (x_array WITH [idx] := satinc(x_array[idx],step)),
      idx+1, step)
  ENDIF;
invs_are_satisfied_i(pc_array: PC_ARRAY, x_array: CLOCK_ARRAY,
  idx: TRAIN_INDEX): BOOLEAN =
    (pc_array[idx] = safe OR
    pc_array[idx] = stop OR
    (pc_array[idx] = appr AND x_array[idx] <= appr_x_max) OR
    (pc_array[idx] = start AND x_array[idx] <= start_x_max) OR
    (pc_array[idx] = cross AND x_array[idx] <= cross_x_max)
  );
invs_are_satisfied(pc_array: PC_ARRAY, x_array: CLOCK_ARRAY,
  idx: TRAIN_INDEX): BOOLEAN =
  IF idx = N THEN invs_are_satisfied_i(pc_array, x_array, N)
  ELSE (invs_are_satisfied_i(pc_array, x_array, idx) AND
    invs_are_satisfied(pc_array, x_array, idx+1))
  ENDIF;
% The function to enqueue a train

enqueue(queue: QUEUE, idx: TRAIN_INDEX): QUEUE =
  queue WITH .count := queue.count + 1
  WITH .train_array[(queue.front+queue.count-1) MOD N + 1] := idx;

% The function which returns the index of the train at the end of the queue

tail(queue: QUEUE): TRAIN_INDEX =
  queue.train_array[(queue.front+queue.count-2) MOD N + 1];

% A function to dequeue the element in the front

dequeue(queue: QUEUE): QUEUE = queue WITH .train_array[queue.front] := 0
  WITH .front := (queue.front MOD N) + 1
  WITH .count := queue.count - 1;

% To check whether the queue is empty

is_queue_empty(queue: QUEUE): BOOLEAN = (queue.count = 0);

% The module defining the timed transition of the system

clock: MODULE =
BEGIN
  GLOBAL gate_pc: GATE_PC
  GLOBAL pc_array: PC_ARRAY
  GLOBAL x_array: CLOCK_ARRAY
  GLOBAL timestep: BOOLEAN
  OUTPUT step: TIME
  GLOBAL queue: QUEUE
  LOCAL urgent_enabled: BOOLEAN

  INITIALIZATION
    gate_pc = free; % Must be done somewhere!
    timestep = TRUE;
    (FORALL (i: TRAIN_INDEX): x_array[i] = 0);

  DEFINITION
urgent_enabled = ((gate_pc = more) AND
  (EXISTS (i: TRAIN_INDEX):
    (pc_array[i] = appr AND x_array[i] <= 10))
  OR
  (NOT(is_queue_empty(queue)) AND
   (gate_pc = free) AND
   (EXISTS (i: TRAIN_INDEX): (pc_array[i] = stop)));

TRANSITION
[
  timeline:
    timestep AND
    invs_are_satisfied(pc_array, x_array', 1) AND
    (urgent_enabled => step' = 0) -->
    step' IN { x: TIME | true };
    x_array' = incr_clocks(x_array,1, step');
    timestep' = FALSE;
]

END;

% ----------------------
% The module defining the behaviour of an individual train.
% Transitions that are independent of gate are put here.

train[i: TRAIN_INDEX]: MODULE =
BEGIN
  GLOBAL x_array: CLOCK_ARRAY
  GLOBAL pc_array: PC_ARRAY
  GLOBAL timestep: BOOLEAN

  INITIALIZATION
    pc_array[i] = safe

  TRANSITION
[
    appr_to_cross:
      NOT(timestep) AND
      pc_array[i] = appr AND
      x_array[i] >= 10 AND
      x_array'[i] <= cross_x_max -->
      pc_array'[i] = cross;
      x_array'[i] = 0;
      timestep' = TRUE

    start_to_cross:
      NOT(timestep) AND
      pc_array[i] = start AND
      x_array[i] >= 10 AND
      x_array'[i] <= cross_x_max -->
      pc_array'[i] = cross;
      x_array'[i] = 0;
      timestep' = TRUE
  ]
x_array[i] >= 7 AND x_array'[i] <= cross_x_max -->
    pc_array'[i] = cross;
    x_array'[i] = 0;
    timestep' = TRUE
]

END;

% ----------------------
% The module defining the synchronous transition appr[i]

appr[i: TRAIN_INDEX]: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL queue: QUEUE
GLOBAL gate_pc: GATE_PC
GLOBAL timestep: BOOLEAN

INITIALIZATION

queue.front = 1;
queue.count = 0;
(FORALL (i: TRAIN_INDEX): queue.train_array[i] = 0)

TRANSITION

[ free_to_occ_and_safe_to_appr:
    % queue is empty and a train is approaching
    NOT(timestep) AND
gate_pc = free AND
classical_array[i] = safe AND
is_queue_empty(queue) AND
x_array'[i] <= appr_x_max -->
gate_pc' = occ;
pc_array'[i] = appr;
x_array'[i] = 0;
queue' = enqueue(queue, i);
timestep' = TRUE
]

[] occ_to_more_and_safe_to_appr:
    % queue is nonempty and a new train is approaching
    NOT(timestep) AND
gate_pc = occ AND
classical_array[i] = safe AND
x_array'[i] <= appr_x_max -->
gate_pc' = more;
pc_array'[i] = appr;
x_array'[i] = 0;
queue' = enqueue(queue, i);
timestep' = TRUE
]
END;

% ----------------------
% The module defining the synchronous transition stop[i]

stop[i: TRAIN_INDEX]: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL queue: QUEUE
GLOBAL gate_pc: GATE_PC
GLOBAL timestep: BOOLEAN

TRANSITION
[
  more_to_occ_and_appr_to_stop:
    NOT(timestep) AND
tail(queue) = i AND
gate_pc = more AND
pc_array[i] = appr AND
x_array[i] <= 10 -->
  pc_array'[i] = stop;
gate_pc' = occ;
timestep' = TRUE
]
END;

% ----------------------
% The module defining the synchronous transition go[i]

go[i: TRAIN_INDEX]: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL queue: QUEUE
GLOBAL gate_pc: GATE_PC
GLOBAL timestep: BOOLEAN

TRANSITION

[free_to_occ_and_stop_to_start:
  NOT(timestep) AND
  queue.count > 0 AND
  queue.train_array[queue.front] = i AND
  gate_pc = free AND
  pc_array[i] = stop AND
  x_array'[i] <= start_x_max -->
  pc_array'[i] = start;
  gate_pc' = occ;
  x_array'[i] = 0;
  timestep' = TRUE
]

END;

% ----------------------
%
% The module defining the synchronous transition leave[i]
%
leave[i: TRAIN_INDEX]: MODULE =
BEGIN

GLOBAL pc_array: PC_ARRAY
GLOBAL x_array: CLOCK_ARRAY
GLOBAL timestep: BOOLEAN
GLOBAL queue: QUEUE
GLOBAL gate_pc: GATE_PC

TRANSITION
[
  occ_to_free_and_cross_to_safe:
  NOT(timestep) AND
  queue.train_array[queue.front] = i AND
  gate_pc = occ AND
  pc_array[i] = cross AND
  x_array[i] >= 3 -->
  gate_pc' = free;
  pc_array'[i] = safe;
  queue' = dequeue(queue);
  timestep' = TRUE
]

END;

% ----------------------
%
% Definitions of the system as an asynchronous composition of
% trains, synchronous transitions, and clock

32
trainsgate: MODULE = 
    ([] (i: TRAIN_INDEX): (train[i] [] appr[i] []
    stop[i] [] leave[i] [] go[i]));

system: MODULE = clock [] trainsgate;

% ----------------------

% Mutual Exclusion property to be checked

mutex: THEOREM
    system |- G(FORALL (i, j: TRAIN_INDEX):
    i/=j => NOT(pc_array[i] = cross AND pc_array[j] = cross));

appr_to_cross: THEOREM
    system |- (FORALL (i: TRAIN_INDEX):
    G((pc_array[i] = appr) => F(pc_array[i] = cross)));
C  Pursuit - Hybrid Automata Model

pursuit: CONTEXT =
BEGIN

% Some typedefinitions
NNREAL: TYPE = {x: REAL | x >= 0};
LOCATION: TYPE = {clockwise, counter, rescued};

% ----------------------
inv_cc(t_val: NNREAL, e_val: NNREAL, p_val: NNREAL): BOOLEAN =
  t_val <= 2 AND
e_val >= 0 AND e_val <= 40 AND
  p_val >= 0 AND p_val <= 40;
% ----------------------

system: MODULE =
BEGIN

LOCAL
  loc: LOCATION,
  timestep: BOOLEAN,
  step: NNREAL,
  t: NNREAL,
  e: NNREAL,
  p: NNREAL,
  new_p : NNREAL

INITIALIZATION
  loc = clockwise;
  timestep = TRUE;
  t = 2;
  e = 20;
  p = 30;

TRANSITION
[
  continous_step:
    timestep = TRUE AND
    step’ >= 0 AND
    inv_cc(e’,t’,p’) AND
    2*new_p’ >= 2*p - step’ AND new_p’ <= p + 6*step’ -->
    t’ = IF loc = rescued THEN t ELSE t + step’ ENDIF;
    e’ = IF loc = rescued THEN e
         ELSEIF loc = clockwise THEN e + 5*step’
              ELSE e - 5*step’ ENDIF;
    p’ = IF loc = rescued THEN p ELSE new_p’ ENDIF;
timestep' = FALSE;

[]
clockwise_clockwise:
timestep = FALSE AND
loc = clockwise AND
t = 2 AND
e /= 0 AND
%inv_cc(t',e',p') AND
5*p - 6*e + 40 < 0 -->
t' = 0;
timestep' = TRUE;

[]
clockwise_rounding_up:
timestep = FALSE AND
loc = clockwise AND
(e = 40 OR p = 40 OR p = 0) -->
e' = IF e = 40 THEN 0 ELSE e ENDIF;
p' = IF p = 40 THEN 0 ELSIF p = 0 THEN 40 ELSE p ENDIF;
timestep' = TRUE;

[]
clockwise_to_rescued:
timestep = FALSE AND
loc = clockwise AND
e = 0 -->
loc' = rescued;
timestep' = TRUE;

[]
clockwise_to_counter:
timestep = FALSE AND
loc = clockwise AND
t = 2 AND
e /= 0 AND
5*p - 6*e + 40 >= 0 -->
loc' = counter;
t' = 0;
timestep' = TRUE;

[]
counter_counter:
timestep = FALSE AND
loc = counter AND
t = 2 AND
e /= 0 AND
5*p - 6*e + 40 >= 0 -->
t' = 0;
timestep' = TRUE;

[]
counter_rounding_up:
timestep = FALSE AND
loc = counter AND
(e = 0 OR p = 40 OR p = 0) -->
\( e' = \text{IF } e = 0 \text{ THEN } 40 \text{ ELSE } e \text{ ENDIF}; \)

\( p' = \text{IF } p = 40 \text{ THEN } 0 \text{ ELSIF } p = 0 \text{ THEN } 40 \text{ ELSE } p \text{ ENDIF}; \)

\( \text{timestep}' = \text{TRUE}; \)

[]

counter_to_rescued:
  timestep = FALSE AND
  loc = counter AND
  e = 0 -->
  loc' = rescued;
  timestep' = TRUE;

[]

counter_to_clockwise:
  timestep = FALSE AND
  loc = counter AND
  t = 2 AND
  e /= 0 AND
  5*p - 6*e + 40 >= 0 -->
  loc' = clockwise;
  t' = 0;
  timestep' = TRUE;

]

END;

% ----------------------

% Properties to be checked

safety: THEOREM
  system |- G(e /= p);

wrong: THEOREM
  system |- G(p = 0);

END