Extending PROMELA and Spin for Real–Time (Extended Abstract)

Stavros Tripakis[1] and Costas Courcoubetis[1] *

Department of Computer Science, University of Crete, Heraklion, Greece, and
Institute of Computer Science, FORTH

Abstract. The efficient representation and manipulation of time information is key to any successful implementation of a verification tool. Two slightly different models of timed automata have been proposed in [Dil89] and [ACD90]. We extend the syntax and semantics of the higher level specification language PROMELA to include constructs and statements based on the above models. We implement this extensions on top of the verification tool Spin.

1 Introduction

PROMELA [Hol90] is a language for the specification of *interactive concurrent* systems. Such systems consist of a finite number of separate *components*, which act independently one from another, and interact through the exchange of *messages* over message *channels*. A large part of these systems, including communication protocols, asynchronous circuits, traffic or flight controllers, and real–time operating systems can be characterized as *real–time* systems. This characterization comes from the following two observations :

1. the correct functioning of those systems depends on the timely coordination of their interacting components ; and,
2. information is available about the *time delays* encountered during the operation of system processes.

The first observation is crucial when trying to ensure that the system meets its requirements. The second one can be used to develop a more efficient system : knowing with certainty some facts about the delays in a system can lead to concluding that a number of behaviors are impossible, and therefore, can be ignored during system design.

Traditional formalisms for temporal reasoning, such as the untimed PROMELA model, deal only with the *qualitative* aspect of time, that is, the *order* of certain system events [2]. However, real–time systems often demand for a *quantitative* aspect of time, that is, taking into consideration the actual distance in time of certain system events [3]. Our motivation to extend PROMELA with real–time capabilities comes from the above reasons. Of the existing models for quantitative temporal reasoning, we consider the *dense* time model as being the best, due to its correctness, expressiveness, and its complexity characteristics (for a complete argumentation of this thesis see [Alu91]). In dense time, an unbounded, however finite, number of events can occur between two successive time moments.

Our verification method consists in considering emptyness of *timed automata* (TA). TA [Dil89, AD90, ACD90] have been introduced for modeling real–time systems, dealing with time as a dense quantity. They can be defined as ordinary *Büchi* automata extended with a finite number of *clocks*. Based on a timed PROMELA specification, we construct the equivalent TA (modulo operational semantics). Then we check if the timed language of this TA is empty. A PROMELA program essentially consists of a collection of components which interact using *asynchronous concurrency* semantics. Optionally, a special component can be specified, (called the *never–claim*) which interacts with the rest of the system *synchronously*, and models the complement of the desired system behavior.

In the absence of the never–claim, the complement of the desired system property is coded explicitly in the components in terms of non–progress conditions. If a claim is specified along with the system,

* Partially supported by the BRA ESPRIT project REACT.

[2] An example of a qualitative time property is : "the green light is never switched on after the red one and before the orange one"

[3] An example of a quantitative time property is : "the orange light will always be switched on at least 5 seconds after the red one, followed in at most 0.5 seconds by the green one"

then we consider their synchronous composition. In either case, the absence of erroneous behaviors can be reduced to a language–emptyness problem.

We use the TA model proposed in [AD90, ACD90], where the clocks are real variables, taking positive or zero values. Each transition of the automaton is annotated with a *constraint* on the clock values, as well as a set of clocks to be *reset*. The constraint acts like a guard, which restricts the executability of a transition (therefore, of a program statement). The clock resets take place instantaneously if the transition succeeds. In [Alu91] it is shown that the class of TA is closed under union and intersection, but not under complement. Also, that the problem of emptyness is solvable in PSPACE, the problems of inclusion or universality being undecidable.

Our work, described in this document, has been, first of all, to extend the syntax and semantics of untimed PROMELA to clocks and time information. We call this extended language *Real–Time–PROMELA* (RT–PROMELA). Next, we have implemented the TA verification procedure on top of *Spin* [Hol94], obtaining *RT–Spin*, a tool for the verification of RT–PROMELA programs. Care has been taken, so that the TA analysis is absolutely compatible with the existing search algorithms used in untimed Spin. Finally, one of our contributions has been the description of a formal semantics of both untimed and RT–PROMELA, based on untimed and timed transition systems, respectively.

The rest of this document is organized as follows. Section 2 gives a brief overview of the theoretical foundations, discussing timed languages and automata simply as a means of modeling real–time systems. In section 3 we present RT–PROMELA. We formally describe its set of operational semantics, in terms of timed transition systems, by referring directly to the global system. We also define the verification problem in the untimed case. In the full paper, we show how one can provide *trace* semantics for individual untimed and RT–PROMELA processes, and derive the semantics of the complete specification in a compositional way. Verification using RT–PROMELA is discussed in section 4. Experimental results are presented in section 5. All missing proofs can be found in the full paper.

2 Timed languages and timed automata

Timed automata have been defined [Alu91], as an extension of Büchi automata [Büc62] to describe timed languages. A *Büchi automaton* (BA) is a nondeterministic finite–state automaton defined by a quintuple (Σ, S, Tr, S_0, F), where :

1. Σ is an input alphabet ;
2. S is a finite set of automaton states ;
3. $S_0 \subseteq S$ is a set of initial states ;
4. $Tr \subseteq S \times \Sigma \times S$ is the transition relation, where a *transition* $tr = (s, \alpha, s') \in Tr$ means that if the automaton is in state s and reads input symbol α it can change state going to s' ;
5. $F \subseteq S$ is a set of *accepting states*.

A *trace* or input word is an infinite sequence $\sigma = \sigma_1\sigma_2...$ of input symbols $\sigma_i \in \Sigma$. A *run* of a BA over σ is the infinite sequence : $r\ :\ s_0 \overset{\sigma_1}{\mapsto} s_1 \overset{\sigma_2}{\mapsto} ...$, such that $s_0 \in S_0$, and $(s_i, \sigma_{i+1}, s_{i+1}) \in Tr$, for all $i = 0, 1, ...$ A run r is said to be *accepting* iff there exists a state $f \in F$ such that f appears infinitely often in r. The *language* $\mathcal{L}(A)$ of a BA A is the set of all traces σ such that A has an accepting run over σ.

2.1 Timed languages

A *timed* word or trace is a pair (σ, τ), where σ is a trace and τ is a *time sequence*, that is, an infinite sequence $\tau_1, \tau_2, ...$ of time values $\tau_i \in \mathsf{R}^+$, satisfying the following constraints :

1. τ increases strictly monotonously : $\tau_i < \tau_{i+1}$;
2. τ is *non–zeno* : for every $t \in \mathsf{R}$, there is some i such that $\tau_i > t$.

Intuitively, if ϵ is a set of possible system *events*, and $\sigma_i \subseteq \epsilon$, then τ_i is interpreted as the moment of occurrence of the set of events σ_i. The second condition implies that there is *progress* of time in the system, so that time does not converge to a bounded value [4]. A *timed language* is a set of timed traces.

[4] This is the characteristic of *zeno* time sequences, for example, $0, 1/2, 3/4, 7/8, ...$

2.2 Timed automata

A TA is a tuple $(\Sigma, S, Tr, S_0, F, C)$, where Σ, S, S_0 and F are as in a Büchi automaton and :

- C is a finite set of clocks ;
- the transition relation is a subset of $S \times \Sigma \times S \times 2^C \times \Phi(C)$, where $\Phi(C)$ is the set of all possible constraints μ, formed by clocks in C. Each constraint is a boolean conjunction of atomic formulas of the form $y \leq k$, $k \leq y$, $x - y \leq k$ and $k \leq x - y$ for two clocks $x, y \in C$, and an integer constant $k \in \mathsf{N}$. The fourth component of a transition is a subset of C, denoting the clocks to be reset.

Intuitively, the operation of a TA A can be described as follows. Given a timed trace (σ, τ), A starts at state $s_0 \in S_0$ at time 0. All the clocks of A are active, initialized to zero, and increase at the same rate. At time τ_1 the symbol σ_1 is read and the automaton takes the transition $tr_0 = (s_0, \sigma_1, s_1, Reset_1, \mu_1)$, assuming that the values of the clocks satisfy μ_1. Then the automaton jumps to s_1 and all clocks that belong in $Reset_1$ are reset to 0. At time τ_2 a new input symbol is read, the next transition is chosen, and so on.

More formally, a run r, denoted by $(\overline{s}, \overline{\nu})$, of a TA over a timed word (σ, τ) is an infinite sequence of the form $r : (s_0, \nu_0) \xrightarrow{\sigma_1, \tau_1} (s_1, \nu_1) \xrightarrow{\sigma_2, \tau_2} ...$ with $s_i \in S$ and ν_i is a clock *valuation*, that is, a vector in $\mathsf{R}^{|C|}$, which represents the values of the clocks at any moment on the run. The following conditions should hold, in order for the run to be valid :

1. (initiation) $s_0 \in S_0$ and $\nu_0(x) = 0, \forall x \in C$;
2. (consecution) for all $i = 1, 2, ...$, there is a transition $(s_{i-1}, \sigma_i, s_i, Reset_i, \mu_i) \in Tr$ such that $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfies μ_i, and $\nu_i(x) = 0, \forall x \in Reset_i$.

If an infinite run exists over (σ, τ), then the timed trace is *timing consistent* with respect to the TA. A timing consistent timed trace is *accepting* if it satisfies the Büchi acceptance conditions described above. The class of languages accepted by a TA is characterized in [Alu91], as the class of timed *regular* languages. Moreover, it is shown that the class is closed under union and intersection (but not under complement). We will be using the notation $A_1 \otimes_s A_2$ to denote the synchronous product of two automata A_1 and A_2. This product is defined to be the TA that accepts the intersection of the languages $\mathcal{L}_1$ and $\mathcal{L}_2$, accepted by A_1 and A_2 respectively.

TA semantics : We will now specify the semantics of a TA in terms of a *timed transition system* (TTS). A *transition system* (TS) is a (possibly infinite) graph, i.e., a pair (S, Tr), where S represents the state space (nodes) of the graph, and $Tr \subseteq S \times S$ its transition relation (edges). A *path* is a sequence of states $s_0, s_1, ...$ such that $(s_i, s_{i+1}) \in Tr, \forall i = 0, 1, ...$ A path $s_0, s_1, ..., s_i$ is *terminating* if there exists no state s_j such that $(s_i, s_j) \in Tr$.

A TTS is a particular TS, (S^τ, Tr^τ), where the states record time information. More precisely, for a TA defined by the tuple $(\Sigma, S, Tr, S_0, F, C)$, we associate the TTS (S^τ, Tr^τ), each state of which has the form (s, ν), where $s \in S$, and ν is a clock valuation. The transition relation Tr^τ contains two types of edges :

1. (Transition successor) If $(s, \nu) \in S^\tau$, and ν satisfies the condition μ of a transition $(s, \sigma, s', Reset, \mu) \in Tr$, then $((s, \nu), (s', [Reset := 0]\nu)) \in Tr^\tau$, where we denote by $[Reset := 0]\nu$ the valuation ν' that assigns 0 to each clock in $Reset$ ($\forall x \in Reset,\ \nu'(x) = 0$ and $\forall x \in C \setminus Reset,\ \nu'(x) = \nu(x)$) ;
2. (Time successor) If $(s, \nu) \in S^\tau$ and $\delta > 0$ is a time increment, then $((s, \nu), (s, \nu + \delta)) \in Tr^\tau$, where $\nu + \delta$ is the valuation that assigns $\nu(x) + \delta$ to each clock in C.

3 RT–PROMELA

The reader can refer to [Hol90, Hol94, Hol95] for a complete presentation of untimed PROMELA. Here we shall present a formal semantics of both the untimed and timed version of the language.

3.1 A review of the PROMELA verification methodology

Language summary : PROMELA programs consist of *processes*, message *channels*, and *variables*. A specification in PROMELA consists in two parts : the *system specification part* (system, for short) and the *property specification part* (the never–claim, which is optional). In the first, a number of process *types* are declared which are then instantiated into *real* processes at run time. A process (usually init) can create other processes (of a certain type) by means of the run statement. Processes execute their statements asynchronously, except in the case of atomic statements, or *rendez–vous* handshakes.

The syntax of the never–claim is just like any other process. However, at most one never claim can be present in the specification. Moreover, it should not *participate* in the execution of the system, but rather *monitor* it. By this we mean that every statement inside a claim is interpreted as a condition, and should not have side effects (i.e., send or receive messages, set global variables, execute run statements etc.). Since the system and the claim operate synchronously, the latter can observe the system's behavior step by step, and catch errors.

The PROMELA semantics : The operational semantics of an untimed PROMELA program will be specified in terms of a TS. First, we shall define the TS W associated with the system specification part of the program. Then, we shall define the TS N associated with the optional never claim. Finally, the TS associated with the whole program will be defined as a synchronous composition of W and N.

As for computer programs, the intuitive notion of a PROMELA system *state* lies in a complete description of what is contained in the memories of each active process, as well as the value of its "program counter". For matters of simplicity, we shall be assuming a known, finite number of processes, which are also active from the start.

Thus, let us consider a PROMELA program, whose system consists of m active processes. Each process P_i has a finite set of *locations* L_i, and two special locations $start_i, end_i \in L_i$, of which the first denotes the starting location of P_i (immediately after its opening bracket {) and the second denotes its final location (immediately before its closing bracket }). Each process also has a set of local variables LV_i. Finally, there exists a set of global variables GV. Assume, for matters of simplicity, that only real variables exist and that no two variables have the same name. We denote by lv_i and gv the vectors belonging to $LV_i \times \mathsf{R}^{|LV_i|}$ and $GV \times \mathsf{R}^{|GV|}$ respectively ; these vectors represent the current values of local and global variables. Let V be the union of all variable sets LV_i and GV. Also, assume that the program has a set of channels $Channels$. Note that the members of this set are simply the names of program's channels, whereas we assume that the channel–*queues* (buffers where the messages are stored) are part of the global or local variables.

We define first the TS $W = (S_W, Tr_W) = P_1 \otimes_a P_2 \otimes_a ... \otimes_a P_m$, associated with the system processes $P_1, P_2, ..., P_m$, where :

- the state space S_W contains states of the form $s = (l_1, l_2, ..., l_m, lv_1, lv_2, ..., lv_m, gv)$, where l_i, lv_i, and gv are as defined above ;
- the transition relation Tr_W contains edges of the form $tr = (s, s')$, based on the statements of the processes. Assume that $s = (l_1, l_2, ..., l_m, lv_1, lv_2, ..., lv_m, gv)$, $s' = (l'_1, l'_2, ..., l'_m, lv'_1, lv'_2, ..., lv'_m, gv')$. Then (s, s') belongs in Tr_W if there exist $i, k \in \{1, ..., m\}$ such that :
  1. either $l'_j = l_j, lv'_j = lv_j$, for all $j \neq i$, and there is an assignment or asynchronous channel operation in process P_i which changes its location from l_i to l'_i, as well as lv_i and/or gv to lv'_i and/or gv' ;
  2. or $l'_j = l_j$, for all $j \neq i$, and $gv' = gv, lv'_j = lv_j$, for all j, and there is a skip or a conditional statement of process P_i which changes its location from l_i to l'_i, and whose condition is satisfied by state s ;
  3. or $l'_j = l_j$, for all $j \notin \{i, k\}$, and $gv' = gv, lv'_j = lv_j$, for all j, and there is a pair of rendez–vous statements A!c of process P_k and A?c of process P_i which change their locations from l_k, l_i to l'_k, l'_i respectively, where c is a real constant ;
  4. or $l'_j = l_j$, for all $j \notin \{i, k\}$, and $lv'_j = lv_j$, for all $j \neq i$, and there is a a pair of rendez–vous statements A!c of process P_k and A?x of process P_i, which change the location of the processes as before, and the value of x in lv'_i or gv' (depending on whether x is a local variable of P_i, or a global one) is c.

Next, we define the TS $N = (S_N, Tr_N)$ associated with the never–claim : S_N contains states of the form (s, l_N, lv_N), where $s \in S_W$ (recall that the claim "watches" the system), l_N is the location of the never claim, and lv_N is a vector describing the values of its local variables. A transition $((s, l_N, lv_N), (s, l'_N, lv'_N))$ belongs in Tr_N if the corresponding statement of the never–claim process is executable according to state s of W.

Finally, we can define the synchronous composition of W and N as follows :

$T = W \otimes_s N = (P_1 \otimes_a P_2 \otimes_a ... \otimes_a P_m) \otimes_s N = (S_W, Tr_W) \otimes_s (S_N, Tr_N) = (S_W \times S_N, Tr)$,

where the transition relation Tr contains the following types of edges :

1. $((s_W, s_N), (s'_W, s'_N)) \in Tr$ iff :
   (a) $(s_W, s'_W) \in Tr_W$,
   (b) $(s_N, s'_N) \in Tr_N$, and
   (c) $s_N = (s_W, l_N, lv_N)$, $s'_N = (s'_W, l'_N, lv'_N)$.
2. $((s_W, s_N), (s_W, s'_N)) \in Tr'$ iff :
   (a) $\nexists s'_W$ s.t. $(s_W, s'_W) \in Tr_W$,
   (b) $(s_N, s'_N) \in Tr_N$, and
   (c) $s_N = (s_W, l_N, lv_N)$, $s'_N = (s_W, l'_N, lv'_N)$.

Case 1 includes transitions defined usually in a synchronous product. As for case 2, it corresponds to PROMELA's *claim–stuttering* semantics. According to these, a sequence which blocks, will be expanded to an infinite sequence, by trying to continue executing sentences of the claim, and keep the system part frozen.

Verification in untimed PROMELA : We will now describe the *correctness criteria* of an untimed PROMELA specification, as they are implied by the various types of analysis performed using the tool Spin. PROMELA statements can be marked with special `end`, `accept`, or `progress` labels. Let us denote by E the set which contains all the final locations end_i mentioned above, as well as all locations in the program labeled with an `end` label. Also, let us denote by F the the set of all locations in the program labeled with an `accept` label.

Then, an untimed PROMELA program generally has two correctness criteria, regarding all possible terminating and infinite behaviors of the TS associated with it. First of all, we consider the case where no never claim is present in the specification. Assume that $W = (S, Tr)$ is the TS corresponding to the program, and let $s_0 = (start_1, start2, ..., start_m, \mathbf{0}, \mathbf{0}, ..., \mathbf{0}, \mathbf{0})$ denote the initial state. The program is :

1. *deadlock free* iff for all terminating paths of W, $s_0, s_1, ..., s_k$, $s_i \in S$, where $s_k = (l_{e_1}, l_{e_2}, ..., l_{e_m}, ...)$, it holds that : $\forall i,\ l_{e_i} \in E$;
2. ω*–correct* iff for all infinite paths of W, $s_0, s_1, ..., s_k, ...$ $s_i \in S$ (where s_0 is as before) there is no location $l \in F$, such that s_l appears in the path infinitely many times.

Intuitively, the first criterion means that all terminating executions of the program are considered as deadlocks, except if the system has reached a *valid* final location. This is true in the case where each process has either reached its final location, or has stopped in a location labeled by `end`. The second criterion means that all execution cycles that pass from a state labeled as `accept` are considered "bad".

If a never claim exists, only infinite behaviors are considered. This is because the never claim is essentially a Büchi automaton, which accepts, by definition, only infinite words. Consequently, there is only one correctness criterion namely, the ω–correctness one, which is defined exactly as in case 2 above, for the composite TS T.

3.2 Time extensions

Syntax. First of all, we add the type `clock` to the declarations of PROMELA variables. Clock variables can be scalar or arrays, and are declared globally [5]. Here is an example of the declaration of clocks :

```
clock x, y, z[5];
```

Our next syntactic modification is to expand each PROMELA statement with an optional *time part*, according to the following grammar rules :

[5] The reason for this is that the clock–space dimension cannot change at run time.

```
         stmnt ::= untimed_stmnt
                 | timed_stmnt
                 ;
   timed_stmnt ::= 'when' '{' constraints '}' untimed_stmnt
                 | 'reset' '{' resets '}' untimed_stmnt
                 | 'when' '{' constraints '}' 'reset' '{' resets '}' untimed_stmnt
                 ;
   constraints ::= one_constraint , constraints
                 ;
one_constraint ::= clock op const
                 | clock op clock '+' const
                 ;
         clock ::= x, y, z ∈ C
                 | x[expr]
                 ;
            op ::= '<'  |  '>'  |  '≤'  |  '≥'  |  '=='
                 ;
        resets ::= x , resets    (x ∈ C)
                 ;
```

The time part corresponds exactly to the μ and $Reset$ parts of a transition of a TA. Here are some examples of timed statements :

when$\{x < 4, x \geq 2\}$ reset$\{x\}$ B!mymesg ;
when$\{z < 1, y \geq 1\}$ reset$\{x, z\}$ a = a*b ;
when$\{x == 1\}$ goto error ;

Notice that the set of inequalities is interpreted as a *conjunction*. For example, the time constraint "when$\{x < 4, x \geq 2\}$" stands for "when$\{x < 4 \wedge x \geq 2\}$". There is no way to express *disjunctions* using a single statement, in other words, there is no way to write : "when$\{x < 4 \vee x \geq 2\}$". Instead, one should use a branching nondeterministic statement, for example :

```
if
:: when{x < 4} reset{x} stmnt_part
:: when{x ≥ 2} reset{x} stmnt_part
fi
```

which is equivalent. The reason for the above restriction will be clear in section 4, where we discuss our verification methodology.

Semantics. As we have done with ordinary PROMELA programs and TSs, we shall establish a mapping between an RT–PROMELA program and a timed TS. Assume, first of all, that (S, Tr) is the TS associated with the untimed part of the RT–PROMELA program. This untimed part can be found by 'stripping" the program from any syntax involving time, that is, clock declarations, and time_parts in the statements. We start from the untimed program, because its possible behaviors include all possible behaviors of the timed one. Indeed, it is clear that no new behaviors can be added as a result of the time extensions, while, on the other hand, the presence of time constraints generally makes a number of behaviors of the untimed program impossible.

Thus, we define the TTS (S^τ, Tr^τ) from (S, Tr). The *extended* state set S^τ contains states of the form (s, ν), where $s \in S$, and ν is a clock valuation. The transition relation Tr^τ contains three types of edges :

1. (Simple transition successor) Assume that $(s, s') \in Tr$ is a transition corresponding to the untimed part of a statement, and that $\mu, Reset$ constitute the time part of the latter. If ν satisfies μ, then $((s, \nu), (s', [Reset := 0]\nu)) \in Tr^\tau$;
2. (Rendez–vous transition successor) Assume that $(s, s') \in Tr$ is a transition corresponding to the untimed parts of a rendez–vous pair of statements. Let $\mu_1, Reset_1$ and $\mu_2, Reset_2$ be the time parts of these statements. If ν satisfies $\mu_1 \wedge \mu_2$, then $((s, \nu), (s', [Reset := 0]\nu)) \in Tr^\tau$, where $Reset = Reset_1 \cup Reset_2$;

3. (Time successor) If $(s, \nu) \in S^\tau$ and $\delta > 0$ is a time delay, then $((s, \nu), (s, \nu + \delta)) \in Tr^\tau$.

The correctness criteria of an RT–PROMELA program are identical to those of an untimed one, described in section 3.1, where instead of the untimed TS, we consider the TTS (S^τ, Tr^τ), introduced above.

4 Verification using RT–PROMELA

Our aim is to reduce the verification of the correctness criteria of RT–PROMELA programs to the problem of TA emptyness. To do this, we shall construct a TA which is equivalent to the given RT–PROMELA specification, that is, the TTS associated with the TA is identical to the TTS associated with the specification. In fact, we shall construct the above TA, based on the TTS (S^τ, Tr^τ) itself. Then we can define correctness criteria for TA, similar to those for TTSs. Once this is done, we can prove that the correctness of the TA constructed from a specification, implies the correctness of the specification, and vice–versa. Finally, we will show how the correctness of the TA can be reduced to the language–emptyness problem. Note that this approach is similar to the one used by [CDCT92].

4.1 Constructing the timed automaton

Assume an RT–PROMELA program P and the associated TTS (S^τ, Tr^τ), as defined in the previous section. Then we define $A_P = (\Sigma, S, Tr, S_0, F, C)$, to be the TA which has the same state space S as the TTS, and :

- its alphabet Σ, contains symbols $\sigma_i = (l_1, l_2, ..., l_m, lv_1, lv_2, ..., lv_m, gv)$, that is, identical to the states of the untimed TS ;
- its initial state set is a singleton $S_0 = \{s_0\}$, where $s_0 = (start_1, start2, ..., start_m, \mathbf{0}, \mathbf{0}, ..., \mathbf{0}, \mathbf{0})$;
- C is the set of declared clocks ;
- F contains all states $(l_{e_1}, l_{e_2}, ..., l_{e_m}, ...)$ where at least for one $i \in \{1, 2, ..., m\}$, location l_i is labeled with an `accept` label ;
- a transition $(s, \sigma, s', Reset, \mu)$ belongs in Tr iff σ is identical to s, and there exists a non–time–successor edge $tr^\tau = ((s, \nu), (s', \nu'))$ of Tr^τ, such that :
  - either tr^τ is a simple edge, its corresponding time part being $\mu, Reset$,
  - or tr^τ is a rendez–vous edge, its corresponding time parts being $\mu_1, Reset_1$ and $\mu_2, Reset_2$, and $Reset = Reset_1 \cup Reset_2$, $\mu = \mu_1 \wedge \mu_2$.

In order to be able to define deadlock–freedom, as well as the usual Büchi–acceptance correctness criterion, we shall need the set $End \subseteq S$ containing all states $(l_{e_1}, l_{e_2}, ..., l_{e_m}, ...)$, where for all $i = 1, 2, ..., m$, either $l_{e_i} = end_i$ or l_{e_i} is labeled with an `end` label.

Recall the notion of a run $(\overline{s}, \overline{\nu})$, defined in section 2.2. Assume now a timed word (σ, τ), and a finite sequence $(s_0, \nu_0) \overset{\sigma_1, \tau_1}{\longmapsto} (s_1, \nu_1) \overset{\sigma_2, \tau_2}{\longmapsto} ... \overset{\sigma_k, \tau_k}{\longmapsto} (s_k, \nu_k)$ over (σ, τ), which satisfies, for all $i = 1, 2, ..., k$, the initiation and consecution conditions of an ordinary run, and for which there exist no τ_l, s_l, ν_l such that $(s_k, \nu_k) \overset{\sigma_{k+1}, \tau_l}{\longmapsto} (s_l, \nu_l)$ satisfies the consecution condition. We call this sequence a *terminating run* over (σ, τ). Such a run is *valid* if $s_k \in End$. We will use these notions in our discussion for correctness right below.

It follows directly from the definitions that A_W has exactly the same semantics (in terms of TTS) as an RT–PROMELA program.

4.2 Verification of RT–PROMELA using TA

We now come to the definition of correctness criteria of TA. A TA $(\Sigma, S, Tr, S_0, F, C)$ is :

1. deadlock free with respect to a set of terminating states $End \subseteq S$ iff all terminating runs are valid with respect to End ;
2. ω–correct iff there is no accepting infinite runs $(s_0, \nu_0) \overset{\sigma_1, \tau_1}{\longmapsto} (s_1, \nu_1) \overset{\sigma_2, \tau_2}{\longmapsto} ...$ over any timed trace (σ, τ).

Now we are ready to prove the following.

Theorem 1. *For an RT–PROMELA program P, and the associated TA A_P :*

- *P is deadlock free iff A_P is deadlock free ; and*
- *P is ω–correct iff A_P is ω–correct.*

Proof. It follows directly from the definitions of A_P and (S^τ, Tr^τ). ∎

So, the problem of checking the correctness of an RT–PROMELA program P is reduced to checking the correctness of A_P. We will now see, how this problem can in turn be reduced to checking the emptyness of the timed language of A_P. First of all, the next corollary follows trivially from theorem 1.

Corollary 2. *For an RT–PROMELA program P, and the associated TA A_P, P is ω–correct iff $\mathcal{L}(A_P) = \emptyset$.*

For deadlock freedom, a slight modification of A_P needs to be made. Let us define $A_P^{dlock} = (\Sigma, S^{dlock}, Tr^{dlock}, S_0, F^{dlock}, C)$, which has exactly the same Σ, S_0, C as A_P, and :

- $S^{dlock} = S \cup \{s^{dlock}\}$, where s^{dlock} is a new state not in S ;
- Tr^{dlock} contains all transitions of Tr, a self–loop transition $s^{dlock}, \sigma, s^{dlock}, \emptyset, true)$, for all alphabet symbols $\sigma =\in \Sigma$, as well as a number of transitions $(s, \sigma, s^{dlock}, \emptyset, true), \forall \sigma =\in \Sigma$, for all states $s \in S \setminus End$;
- $F^{dlock} = \{s^{dlock}\}$.

Intuitively, A_P^{dlock} accepts all timed traces which lead to an invalid end–state, by moving to its unique accepting state, s^{dlock}, and getting trapped into an infinite loop. It is now easy to prove the following lemma.

Lemma 3. *For an RT–PROMELA program P, and the associated TA A_P^{dlock}, P is deadlock free iff $\mathcal{L}(A_P^{dlock}) = \emptyset$.*

Proof. It follows directly from the definitions. ∎

Once we have constructed A_P and A_P^{dlock}, our problem is to check the emptyness of their timed languages. Nevertheless, for this purpose, it suffices to consider the *untimed* projection of these languages. The untimed projection of a timed language $\mathcal{L}$ is defined to be : $Untime(\mathcal{L}) = \{\sigma \mid \exists \tau \ s.t. \ (\sigma, \tau) \in \mathcal{L}\}$. Indeed, as it is shown in [Alu91], the following holds : $Untime(\mathcal{L}) = \emptyset$ iff $\mathcal{L} = \emptyset$.

The idea is to construct, based on the TA A, a normal BA that accepts exactly $Untime(\mathcal{L}(A))$. This automaton will also have an extended state space, where each state will contain, apart from the state of A, the set of all possible clock valuations. Since time is viewed as a dense quantity, that is, the clocks have real values, the above set is, in general, infinite. To solve this problem, the valuation space $\mathsf{R}^{|C|}$ is partitioned into a finite number of *equivalence classes*, $\alpha \subseteq \mathsf{R}^{|C|}$. Two members ν and ν' of a class α are equivalent in the sense that, if ν belongs to an accepting run $(s_0, \nu_0) \stackrel{\sigma_1,\tau_1}{\longmapsto} (s_1, \nu_1) \stackrel{\sigma_2,\tau_2}{\longmapsto} ... \stackrel{\sigma_i,\tau_i}{\longmapsto} (s_i, \nu) \stackrel{\sigma_{i+1},\tau_{i+1}}{\longmapsto} (s_{i+1}, \nu_{i+1})...$, then it can be substituted by ν', which gives another accepting run $(s_0, \nu_0) \stackrel{\sigma_1,\tau_1}{\longmapsto} (s_1, \nu_1) \stackrel{\sigma_2,\tau_2}{\longmapsto} ... \stackrel{\sigma_i,\tau_i}{\longmapsto} (s_i, \nu') \stackrel{\sigma_{i+1},\tau_{i+1}}{\longmapsto} (s_{i+1}, \nu'_{i+1})...$, so that the untimed projections of the two runs are the same. The details of the definitions of equivalence classes can be found in [Alu91], as well as in the full paper. The following theorem comes from [Alu91].

Theorem 4. *Given a TA $A = (\Sigma, S, Tr, S_0, F, C)$, there exists a Büchi automaton over Σ, which accepts $Untime(\mathcal{L}(A))$.*

Intuitively, a state of the BA U which accepts $Untime(\mathcal{L}(A))$ will be a pair (s, α). The first step in the construction of U, is to costruct an automaton without Büchi acceptance conditions, called the *region automaton* $R(A)$. This mimics the runs of A, keeping at the same time information on the possible range of values of the system's clocks. The details of the construction are presented in the full paper. The problem of checking the emptyness of U is reduced to a reachability analysis over the corresponding TTS, say G_U. In other words, a depth–first–search on G_U has to be performed, in search for loops. We shall call G_U the *state graph* of U. Notice that, since S is finite, and the number of equivalence classes α is finite (depending on the number of clocks and their constraints), the state space of U, is also finite, and so is the state graph G_U. Therefore, termination is ensured.

4.3 Checking emptyness efficiently

As it can be made clear from our previous discussion, the construction of the region automaton is key to any efficient verification method, based on reachability analysis. Some basic technics for inexpensive ways to construct and search the region automaton are presented in the following section. Here, we shall refer to one aspect (perhaps the most important) of the problem, namely, how to represent time information with the least possible cost.

Assume that $R(A)$ is the region automaton that accepts $Untime(\mathcal{L}(A))$, and $G_{R(A)}$ its state graph mentioned above. As in $R(A)$, a state of the graph $G_{R(A)}$ is a pair (s, α), where α represents time information in the form of an equivalence class. This implies that during the search, each "untimed" state s of the system can be visited up to $|\mathcal{C}_\alpha|$ times, where $\mathcal{C}_\alpha$ denotes the set of equivalence classes of the specification in question. Since this number can be quite large, we would like to be able to do the analysis in terms of *unions* of equivalence classes. We call such a union *clock region*, and will be denoting it by CR. The next problem is to find an efficient way to represent clock regions.

A very popular representation is using *difference bounds matrices* (DBMs) [Dil89]. DBMs have the advantage of being inexpensive as far as storage is concerned. Moreover, they are simple and require low–cost operations. The reader can refer to [Dil89] for the relative background. Briefly, a DBM is a square matrix which describes a very simple system of linear inequalities, of the same form as time constraints, that is, x op k, or x op $y + k$, where op $\in \{<, >, \leq, \geq, =\}$, and k is a positive integer constant. Assuming the dimension of a matrix D to be $n \times n$, the set of vectors $\nu \in \mathsf{R}^n$ which satisfy the corresponding inequalities will be denoted $Q(D)$. This set is convex. Then, the idea is to represent a clock region CR by a DBM D, so that $Q(D) = CR$. At each step during the reachability analysis, a new DBM is computed by transforming the old one. For this, a small number of relatively low–cost operations are used, which shall be now briefly described.

The *intersection* of two DBMs corresponds to the intersection of the regions the DBMs represent. The *time–elapse* transformation yields a new DBM which contains all time–successor valuations of the old one, by letting an arbitrary amount of time (possibly zero) elapse. The *clocks–reset* transformation yields a new DBM where some clocks are reset to zero.

In general, more than one DBMs can be used to represent the same set of clock valuations. This is due to the fact that the bounds found in certain inequalities are not "strict" enough. Nevertheless, it is possible to obtain the *canonical* form of a DBM, which is its unique, "minimal" representative. This is done by the process of DBM *canonicalization*. Let $\mathrm{cf}(D)$ denote the canonical form of a DBM D, and D_1, D_2 be two different matrices. The following holds :

$$Q(D_1) = Q(D_2) \Leftrightarrow \mathrm{cf}(D_1) = \mathrm{cf}(D_2).$$

The use of canonical form reduces the test for equality of two matrices to a test for the equality of their canonical forms. This is in turn reduced, at the implementation level, to a test for pointer equality, since all DBMs are usually stored in a hashing table. The rest of the DBM operations are also simplified by the use of canonical forms.

During the series of transformations, it is possible that the resulting DBM does not "cover" exactly a clock region. Indeed, a clock region is a union of equivalence classes, which is not always convex, while the region represented by a matrix always is. However, the matrix can be enlarged to include as many points of the region not contained in the initial matrix as possible, resulting in a canonical representation [6]. The whole process is called *maximization*. For a DBM D, and the maximized one, $max(D)$ the following property holds for all other DBM D' :

$$(\forall \alpha\ (\alpha \cap Q(D) \neq \emptyset \Leftrightarrow \alpha \cap Q(D') \neq \emptyset)) \Rightarrow Q(D') \subseteq Q(max(D)),$$

where α, as usual, denotes an equivalence class.

The precise definitions of the above transformations are presented in the full paper.

To prove the correctness of our approach, let us define another automaton, called the *DBM automaton*, denoted A_{DBM}. This will play the same role as the region automaton of A, that is, it will follow exactly the same runs as A does, keeping track of the possible clock positions at each step.

For a TA $A = (\Sigma, S, Tr, S_0, F, C)$, define $A_{DBM} = (\Sigma, S', Tr', S'_0)$ such that :

– the states of A_{DBM} are of the form (s, D), where $s \in S$ and D is a DBM ;

[6] It is not wrong to add these extra points, since each one of them is equivalent with at least one point in the matrix, thus satisfies exactly the same properties regarding the evolution of the system in time.

- the initial states of A_{DBM} are of the form (s_0, D_0), where $s_0 \in S_0$ and D_0 is the DBM such that $Q(D_0) = \{\nu_0\}$ and ν_0 is the valuation that assigns 0 to all clocks ;
- A_{DBM} has a transition $((s, D), \sigma, (s', D'))$ iff there is a transition $(s, \sigma, s', Reset, \mu) \in Tr$, and D' is obtained by D with the following sequence of DBM transformations :

$$D \overset{\delta}{\longmapsto} D^{\delta} \overset{\mu}{\longmapsto} D^{\mu} \overset{\mathrm{cf}}{\longmapsto} D^{cf} \overset{[Reset:=0]}{\longmapsto} D^0 \overset{max}{\longmapsto} D^{max} = D'.$$

In the above sequence, $\overset{\delta}{\longmapsto}$ represents the time–elapse transformation, that is, $\forall \nu \in Q(D), \delta \geq 0,\ \nu + \delta \in Q(D^{\delta})$. $\overset{\mu}{\longmapsto}$ represents the intersection with the constraint μ, that is, $\forall \nu \in Q(D^{\mu})$, ν satisfies μ. $\overset{[Reset:=0]}{\longmapsto}$ represents the clock resets, that is, $\forall \nu \in Q(D^0), x \in Reset,\ \nu(x) = 0$. Finally, $\overset{max}{\longmapsto}$ represents the maximization process. Intuitively, the whole series of transformations corresponds to the fact that, being in a state, the system lets the time pass first (this can be zero time) and then executes a statement/transition instantaneously, moving to another state. In order for the transition to be taken, the time constraints must be satisfied. At the same moment, a number (possibly zero) of clocks are reset to zero.

Not all paths which are discovered during the reachability analysis are valid. Indeed the presence of time gives meaning only to those infinite executions for which time progresses without bound (recall non–zeno timed traces, defined in section 2.1). The notion of time *progressiveness* for runs of the DBM automaton is similar to the one for runs of the region automaton $R(A)$ (see the full paper, or [Alu91]).

A run of A_{DBM}, $r = (s_0, D_0) \overset{\sigma_1}{\mapsto} (s_1, D_1) \overset{\sigma_2}{\mapsto} ...$ over a trace σ, is progressive iff for each clock $x \in C$:

1. there are infinitely many i's such that D_i satisfies $(x = 0) \vee (x > c_x)$, where c_x is the maximum constant that appears in an inequality of the form x op c_x in the specification ;
2. there are infinitely many j's such that D_j satisfies $x > 0$.

Proposition 5. *For the trace σ, there exists a progressive run $r\ :\ (s_0, \alpha_0) \overset{\sigma_1}{\mapsto} (s_1, \alpha_1) \overset{\sigma_2}{\mapsto} ...$ of $R(A)$ over σ iff there exists a progressive run $r'\ :\ (s_0, D_0) \overset{\sigma_1}{\mapsto} (s_1, D_1) \overset{\sigma_2}{\mapsto} ...$ of A_{DBM} over σ.*

5 Examples

We have implemented the method described above on top of the tool Spin, developed for the validation of concurrent systems [Hol90], by G. J. Holzmann. We have extended Spin to RT–Spin, which performs reachability analysis on the DBM automaton, using as input an RT–PROMELA program.

We have tested our implementation using a number of examples. We now present three of them. The first one models a simple system of three processes, representing, respectively, a train, a gate, and a controller. The second is a real–time mutual–exclusion protocol, due to Fischer [Lam87]. Both these examples have been taken from [ACD+92a]. The third has to do with a general–purpose ATM switch [Sid91]. It has been taken from [Lam93], where it has been treated using a different model.

Generally, the systems consist of a number of components, modeled as TA. RT–PROMELA offers the possibility to use local and global variables, as well as channels. We take advantage of this capability, and we end up with less components than those described in the original models. For example, we do not need a special automaton to model the global variable in the mutual–exclusion protocol. Components synchronize their actions through shared events. Associated with every component is a number of event symbols, and an event can occur provided it is enabled in every automaton whose alphabet includes the event. Whenever necessary, synchronization in RT–PROMELA is done using rendez–vous. We now describe the three examples in detail.

5.1 Modeling the systems using RT–PROMELA

Train, Gate, Controller (TGC) : This example deals with an automatic controller that opens and closes a gate at a railway track intersection (see figure 1). Whenever the train enters the intersection it sends an `approach` signal at least two minutes in advance to the controller. The controller also detects the train leaving the intersection and this event occurs within five minutes after it started its approach. The gate responds to `lower` and `raise` commands by moving `down` and `up` respectively within certain time bounds. The controller sends a `lower` command to the gate exactly one minute

Fig. 1. Train, Gate, Controller

Fig. 2. Timed mutual exclusion

after receiving an approach signal from the train. It commands the gate to raise within one minute of the train's exit from the intersection.

The purpose of the verification is to ensure the following safety property [7] : whenever the gate goes down, it is moved back up within a certain upper time bound K. Notice that this implies that the gate *will* eventually come up again. Although this is not immediate from the above property, *liveness* conditions that are associated with each automaton ensure that in every infinite trace process Gate passes infinitely often from state q0, therefore executes infinitely often the transition $q3 \longrightarrow q0$ which means that the gate is up. Returning to the safety property, the automaton Monitor models precisely the negation of it, as was explained in section 2.2. It can be easily observed that the property is satisfied whenever the integer constant K is greater than 6.

In order to model process synchronization in RT–PROMELA, we used zero–length channels.

[7] A *safety* property can be formulated as "never will...". For example "never will processes 1 and 2 be found at their critical sections at the same time"

Fig. 3. Two ATM switches

Timed Mutual Exclusion : In this protocol, there exist n identical processes, as shown in figure 2. Each process is labeled with a unique identifier $i \in 1..n$. It is initially idle, but at any time may begin executing the protocol provided the value of a global variable x is 0. It then delays for up to Δ_B seconds before assigning the value i to x. It may enter its critical section within δ_c seconds provided the value of x is still i. Upon leaving its critical section, it reinitializes x to zero. There is another global variable, `crit`, used to keep count of the number of processes in the critical section. The auto–increment (auto–decrement) of the variable is done simultaneously with the test (reset of x to zero). This is modeled in RT–PROMELA using `atomic` sequences. As usual, we need to verify that no two processes are ever in their critical sections at the same time. The property is satisfied whenever $\Delta_B > \delta_c$.

Verifying the round–trip delay of an ATM switch : An *ATM switch* [Sid91, KSC91] is a chip used as part of the *Asynchronous Transfer Mode* network protocol for *Broadband Interactive Services Data Networks* (B–ISDN). It consists of four input and four output links, each one of 400 Mbits/sec bandwidth. In ATM, information is transferred in *cells* of fixed length (53 bytes). These cells are routed using *virtual circuits* [8] (VCs), which have different priorities. There exists a *flow–control* mechanism, where a special control packet, called *token*, is used to signal to a sender, that the receiver is ready

[8] There exist also *virtual paths*, which are collections of VCs, but will be ignored, since the chip itself cannot distinguish them from VCs

to accept a new high priority cell. For this purpose, each chip has a *flow–control–buffer*, which stores the incoming tokens, as well as a *cell–buffer*, used to store the incoming high priority cells.

In our simplified example, we assume two adjucent chips, say, A and B (see figure 3). We are interested in computing the *round–trip* delay. This is defined in [Sid91, KSC91] as "the delay between the start of two consecutive transmissions of cells of the highest priority", since the chips deal with VCs of different priorities. We make two assumptions :

1. Chip A has always a high priority cell waiting to be transmitted to B.
2. A high priority cell which is sent from B to C is not flow–controlled by C, that is, chip C is always ready to receive it from B.

The first hypothesis allows us to ignore the cell–buffer of chip A, while the second allows us to ignore the flow–control–buffer of B. The timing assumptions of the system are the following :

1. each chip operates on a cycle of 54 clock *ticks*, that is, between any two packet transmissions, there is a delay of exactly 54 ticks ;
2. the delay between the selection of the next packet to be send and its transmission is between 4 and 10 ticks ;
3. the transmission of a token takes between 11 and 33 ticks ;
4. the transmission of a high priority cell takes exactly 3 ticks ;

Based on these assumptions, we prove that the round–trip delay is never greater than 108 clock ticks, while it cannot be less, of course, than 54 ticks. In other words, under the above conditions, at most one low priority packet gets transmitted between any two successive transmissions of high priority cells.

The specification of the mutual–exclusion protocol can be found in the appendix, as a sample RT–PROMELA program.

5.2 Proving safety properties

We used three different methods to verify the properties of the systems above. In the case of TGC, the monitor process moves to an error state marked with an `accept` label, where it stays forever. We ran the validator with the "–a" option to search for acceptance cycles. Notice that in this simple case, maximization wasn't necessary. The results are shown in table 5.2.

The specification of fischer's mutual–exclusion protocol includes a never–claim. This monitors the system and announces an error if it finds out that more than one processes are in the critical section. We verified the correctness of the protocol when $\Delta_B = 1$, $\delta_c = 2$ for up to 4 processes, while in the case of 5 the validator refused to terminate. On the other hand, the erroneous case ($\Delta_B = 2$, $\delta_c = 1$) is very little affected by the size of the problem, since the error is found and announced early on. We managed to ran the wrong case for more than 23 processes. The results of this example are shown in table 5.2.

Finally, for the ATM switch, we make use of a clock RT which keeps count of the round–trip delay, and test the value of the clock each time a new high–priority cell is sent. If RT is between 54 and 108, it is reset to zero and the system continues normally. If the clock is strictly less than 54 or greater then 108, the error is announced. The performance results are shown in table 5.2.

Time is measured in seconds, and memory in megabytes.

K	states	transitions	DBMs	time	memory
5 (erroneous)	26	33	21	0.2	0.8
7 (correct)	32	34	23	0.1	0.9

Table 1. Results of train, gate, controller

N	specification	states	transitions	DBMs	time	memory
23	$\Delta_B = 2,\ \delta_c = 1$ (erroneous)	327	352	172	10.1	1.7
5	$\Delta_B = 2,\ \delta_c = 1$ (erroneous)	93	100	46	0.1	0.9
	$\Delta_B = 1,\ \delta_c = 2$ (correct)	825,610 + ?	1,269,690 + ?	32,265 + ?	1,724 + ?	6 + ?
4	$\Delta_B = 2,\ \delta_c = 1$ (erroneous)	80	86	39	0.1	0.8
	$\Delta_B = 1,\ \delta_c = 2$ (correct)	28,254	43,490	1,869	37.5	2.2
3	$\Delta_B = 2,\ \delta_c = 1$ (erroneous)	67	72	32	0.1	0.8
	$\Delta_B = 1,\ \delta_c = 2$ (correct)	974	1,385	127	0.4	1.2
2	$\Delta_B = 2,\ \delta_c = 1$ (erroneous)	54	58	17	0.1	0.8
	$\Delta_B = 1,\ \delta_c = 2$ (correct)	54	70	13	0.1	0.8

Table 2. Results of Fischer's mutual–exclusion protocol

states	transitions	DBMs	time	memory
435	619	510	2.6	1.5

Table 3. Results of round–trip delay verification

6 Conclusions

We have presented the theory and practice of the extensions made to PROMELA, to include real–time semantics. The extended language, RT–PROMELA, allows for a special kind of global variables, which represent the clocks of the system. The statements of the language can contain simple linear constraints which restrict the possible values of a clock, or the relative values between two clocks. This changes the executability semantics of a statement, which can which can be executed only if, in addition to the restrictions imposed by standard PROMELA, the constraints do not come against the current state of the clocks. Apart from constraints, clocks can also be reset simultaneously with the execution of a statement.

The semantics of a specification in RT–PROMELA are given in terms of timed transition systems. The problem of verification is reduced to checking if the set of all possible valid paths (that is, the language of the system) is empty.

This time model permits the specification of a large class of real–time systems. We illustrate its power by three examples, which have been already considered in the bibliography, thus, allow for comparisons.

Putting clocks in an untimed specification usually increases the size of the state space. There are cases where timing constraints restrict the number of possible behaviors of the system, thus creating

model	states	transitions	DBMs	time	memory
untimed	1,298	3,314	—	0.2	1
timed	19,197	47,180	1,225	14	2.3

Table 4. Results of leader–election protocol

less states than the untimed model. However, most of the times, the size is significantly increased, up to one or two orders of magnitude, as it is shown in table 5.2. There, we show the results obtained by an exhaustive depth–first search performed on an untimed and a timed model of the leader–election protocol [DKR82].

Therefore, future work mainly concerns the research for methods of *reduction*. The size of larger examples makes their analysis prohibitive. The new version of untimed Spin implements the *partial–order* method presented in [Pel94, HP94]. It would be very interesting to see whether this reduction preserves time properties, and under which conditions. Apart from the above, older methods for on–the–fly *minimization* of the state space exist [ACD+92b] and should be also tried out.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 414–425, Philadelphia, June 1990.

[ACD+92a] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *RTSS 1992, proceedings*, 1992.

[ACD+92b] R. Alur, C. Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. Minimization of timed transition systems. In *CONCUR 1992, proceedings*. Lecture Notes in Computer Science, Springer-Verlag, 1992.

[AD90] Rajeev Alur and David Dill. Automata for Modeling Real-Time Systems. In *Automata, Languages and Programming : 17th Annual Colloquium*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335, 1990. Warwick University, July 16-20.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, 1991.

[Büc62] R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Congr. Logic, Method and Philos. Sciences*. Stanford U. Press, 1962.

[CDCT92] C. Courcoubetis, D. Dill, M. Chatzaki, and P. Tzounakis. Verification with real-time COSPAN. In *Proceedings of the Fourth Workshop on Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 1992.

[Dil89] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proc. Workshop on Computer Aided Verification, CAV89*, Grenoble, June 1989. Lecture Notes in Computer Science, Springer-Verlag.

[DKR82] Dolev, Klawe, and Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *J. of Algs*, 3:245–260, 1982.

[Hol90] G.J. Holzmann. *Design and Validation of Protocols*. Prentice-Hall, 1990.

[Hol94] G.J. Holzmann. Basic spin manual. Technical report, AT&T, Bell Laboratories, 1994.

[Hol95] G.J. Holzmann. What's new in spin. Technical report, AT&T, Bell Laboratories, 1995.

[HP94] G.J. Holzmann and Doron A. Peled. An improvement in formal verification. In *Proceedings of the 7th International Conference on Formal Description Techniques, FORTE94*, Berne, Switcherland, october 1994.

[KSC91] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis. Weighted round-robin cell multiplexing in a general purpose ATM switch chip. *IEEE JSAC*, 9(8):1265–1279, 1991.

[Lam87] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[Lam93] N. Lambrogeorgos. Verification of real-time systems: a case study of discrete and dense time models, 1993. Available only in greek.

[Pel94] Doron A. Peled. Combining partial order reductions with on–the–fly model checking. In *Proceedings of the 6th International Conference on Comptuter Aided Verification, CAV94*, Stanford, California, june 1994.

[Sid91] S. Sidiropoulos. A general purpose ATM switch. architecture and feasibility study, 1991.

Acknowledgments

We wish to thank Gerard Holzmann for providing numerous answers regarding the implementation, as well as for attentively reading the paper and making useful remarks.

Appendix : mutual exclusion in RT–PROMELA

```
#define N 5 /* number of processes */
#define deltaB 1
#define deltaC 2
#define ErRoR assert(0)
clock y[N];
int x, crit;

proctype P ( byte id )
{
    do ::
        reset{y[id]} x==0 ->
        when{y[id]<deltaB} reset{y[id]} x=id+1 ->
        atomic{ when{y[id]>deltaC} x==id+1; crit++; } ->
        atomic{ x=0; crit--; }
    od
}

never{
    skip -> /* to let the processes be activated */
    do
        :: crit>1 -> ErRoR
        :: else
    od
}

init {
    byte proc;
    atomic {
        crit = 0;
        proc = 1;
        do
        :: proc ≤ N ->
                run P ( proc%N );
                proc = proc+1
        :: proc > N -> break
        od
    }
}
```