Integrating Real Time into Spin: A Prototype Implementation

Category: Research Paper

Dragan Bošnački* Dennis Dams‡

Abstract

We present a discrete time extension of Promela, a high level modelling language for specification of concurrent systems, and the associated Spin model checker. Our implementation is fully compatible with Spin's partial order reduction algorithm, which is indeed one of its main strengths.

The real time package is for most part orthogonal to the other features of the tool, resulting in a modular extension. We have evaluated it by several experimental results, with encouraging results.

Keywords: verification, real-time systems, model checking, Spin, partial order reduction, Fischer's protocol, Bounded Retransmission protocol.

1 Introduction

Promela is a high level modelling language for specification of concurrent systems. The models written in Promela serve as input to the *Spin* software package for their automated verification.

The time ordering of actions in a Promela program is implicit and depends on the (fixed) sequential composition of statements within each one of the component processes, as well as on the (unspecified) interleaving of statements from different processes. This time relation is only *qualitative*, meaning that we do not know the exact time interval that will elapse between two events. This can be a shortcoming when systems are to be verified whose correct functioning depends on timing parameters. Many such examples can be found among communication protocols that have to deal with unreliable transport media, where the duration of timeout intervals is important.

*Contact author. Address: Dept. of Math. & Computing Science, Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven, The Netherlands. e-mail: `dragan@win.tue.nl`. On leave from the Institute of Informatics, Faculty of Natural Sciences and Mathematics, University "Sts. Cyril and Methodius", Skopje, Macedonia. Supported by EC ESPRIT LTR Project No. 23498 (VIRES).

‡Dept. of Math. & Computing Science, Eindhoven University of Technology, The Netherlands. Supported by the Netherlands Computer Science Research Foundation (SION).

In this paper, we introduce an extension to Promela that allows to quantify the time elapse between events, by specifying the *time slice* in which they occur. We describe a prototype implementation that integrates this extension into the Spin tool. Of particular concern is the compatibility of such an extension with the *partial order reduction* algorithm, which is an approach to alleviate the state-space explosion inherent in model checking, and indeed one of Spin's main strengths. We prove that Spin's partial order algorithm remains correct under the new timing constructs, and conduct a number of experiments that demonstrate its effectiveness.

The prototype implementation described in this paper grew out of an earlier implementation of discrete time in Promela and Spin [4] instigated by process algebra ACP [3] and clocked transition systems [15]. There, the timing constructs from the process algebra ACPdrt were modelled by macro definitions entirely on the level of Promela. An advantage of such an approach is that the real-time package thus obtained is orthogonal to Spin: when needed, it can be incorporated as a simple header-file inclusion, for the current and future versions of Spin. Furthermore, the resulting real-time extension is easily moderated, allowing to gain experience with several alternative syntactical constructs and semantic models of time, and also to study the interaction with Spin's other features, most notably its partial order reduction algorithm. An obvious drawback of such a "high-level" solution is its inefficiency, which is to be avoided particularly in the case of a model checker. The real-time extension of Promela described in the current paper still uses macro definitions for certain constructs. However, the operation that occurs most frequently, namely the advance of time ("tick"), has been implemented on a lower level, in the source code of the Spin tool.

Eventually, we envisage a more complete integration of the real-time package into Spin, not only regarding the level of implementation, but the integration of real time and partial order reduction as well. Also, the underlying mathematical model will be changed from discrete to dense time. While this complicates the implementation, it is commonly accepted (c.f. [1]) that the resulting model-checking algorithms for dense time have comparable computational complexity. Having said this, we do stress that also discrete-time models are sufficient for a broad range of practical applications [11].

A closely related development is the RT-Spin package of [22]. Although that extension of Spin is more general in that it can model Timed Automata ([1]), with real-valued clocks, it has a number of drawbacks. The most important one is that it is not compatible with the partial order reduction algorithm. Furthermore, Timed Automata lack a notion of urgency; instead, the fact that a transition with a deadline must be taken on time, has to be modelled explicitly in RT-Spin. Also, the package has not been kept up-to-date with Spin versions later than Version 2.0.

The extension of partial order reduction techniques to apply to real-time systems has recently been studied in [18] and [17]. Other model checkers that cover timed behaviour are, e.g., UPPAAL ([14]), COSPAN ([2]), and HyTech ([10]). The latter indeed is able to handle the more general class of *linear hybrid systems*.

2 The Spin Model Checker

Promela and Spin have been developed for the analysis and verification of communication protocols. The language syntax is derived from C, but also uses the denotations for communications from Hoare's CSP and control flow statements based on Dijkstra's guarded commands. The full presentation of Promela and Spin is beyond the scope of this paper. We suggest [12] as a reference to the interested reader; here, we only give a brief overview of Spin's verification capabilities.

In Promela, system components are modeled as *processes* that can communicate via *channels* either by buffered message exchanges or rendez-vous operations, and also through shared memory represented as *global variables*. The execution of statements is asynchronous and interleaved, which means that in every step only one *enabled* action is performed, without any assumptions of the relative speed of process executions.

Given as input a Promela model, Spin generates a C program that performs a verification of the system by scanning the state space using a depth-first search (DFS) algorithm. This way, both *safety* properties such as absence of deadlock, unspecified message receptions, invalid end states and assertions can be checked, as well as *liveness* properties such as non-progress cycles and eventual reception of messages. The so-called *never claims*, which are best seen as monitoring processes that run in lock step with the rest of the system, are the most general way to express properties in Spin. Being Büchi Automata, they can express arbitrary omega-regular properties. Spin provides an automatic translator from formulae in linear-time temporal logic (LTL) to never claims. When errors are reported, the trace of actions leading to an invalid state or cycle is saved, so that the erroneous sequence can be replayed as a guided simulation.

For large state spaces methods, Spin features state-vector compressing, partial-order reduction and bit-state hashing.

2.1 Partial Order Reduction

Partial order reduction (POR) is a technique to cope with the state explosion problem in model-checking. We give a brief introduction to the POR algorithm that is used in Spin, rephrasing some definitions from [13].

The basic idea of the reduction is to restrict the part of the state space that is explored by the DFS, in such a way that the properties of interest are preserved. To this purpose, the independence of the checked property from the possible interleavings of statements is exploited. More specifically, two statements a,b are allowed to be permuted precisely then, if for all sequences v,w of statements: if $vabw$ (where juxtaposition denotes concatenation) is an accepted behaviour, then $vbaw$ is an accepted behaviour as well. In practice, sufficient conditions for such permutability are used that can be checked locally, i.e., in a state. For this, a notion of "concurrency" of statements is used that captures the idea that transitions are contributed by different, concurrently executing processes of the system.

The semantics of a Promela program can be represented as a *labeled transition system* (LTS). An LTS is a triple (S, s_0, T), where S is a finite set of states, s_0 is a distinguished initial state, and $T \subseteq S \times S$ is a set of transitions. Every transition in T is the result of executing a statement in some process of the Promela program. We introduce a function $Label$ that maps each transition to the corresponding statement. For a statement a, $Pid(a)$ denotes the process to which a belongs and $En(a)$ denotes the set of (global) states in which a is enabled. For $q \in En(a)$, $a(q)$ is state which is reached by executing a in state q.

Concurrent statements (i.e. statements with different Pids) may still influence each other's enabledness, whence it may not be correct to only consider one particular order of execution from some state. The following notion of *independence* defines the absence of such mutual influence. Intuitively, two statements are independent if in every state where they are both enabled, they cannot disable each other, and are commutative, i.e., the order of their execution makes no difference to the resulting state.

Definition 2.1.1 *The statements a and b are* independent *iff for all states q such that $q \in En(a)$ and $q \in En(b)$,*

- $a(q) \in En(b)$ *and* $b(q) \in En(a)$, *and*
- $a(b(q)) = b(a(q))$.

Statements that are not independent are called dependent.

Note that a and b are trivially independent if $En(a) \cap En(b) = \emptyset$. An example of independent statements are assignments to or readings from local variables, executed by two distinct processes.

Another reason why it may not be correct to only consider only one particular order of execution from state s of two concurrent statements a and b is that the difference between the intermediate states $a(s)$ and $b(s)$ may be observable in the sense that it influences the property to be checked. For a given proposition p that occurs in the property (an LTL formula), and a state s, let $p(s)$ denote the boolean value of the proposition p in the state s. Then, a is *invisible* iff for all propositions p in the property and all states $s \in En(a)$, we have $p(s) = p(a(s))$. a is said to be *safe* if it is invisible and independent from any other statement b for which $Pid(b) \neq Pid(a)$.

The reduction of the search space is now effected during the DFS, by limiting the search from a state s to a subset of the statements that are enabled in s, the so-called *ample set*. Such an ample set is formed in the following way: If there is a process which has only safe statements enabled and all those transitions lead to a state which is not on the DFS stack, then the ample set consists of all the statements from this process only. Otherwise, the ample set consists of all enabled statements in s. It can be proven [13, 19] that the reduced graph obtained in this way preserves the properties of the original LTS, stated as an LTL formula. The condition that all transitions from the ample set must end out of the DFS stack, the so-called "cycle proviso", ensures that a statement that it is constantly enabled, cannot be "forgotten" by leaving it outside the ample set in a cycle of transitions.

While the cycle proviso is clearly locally checkable during a DFS, the condition that an enabled statement is safe is not, as the definition of safety requires independence from *any* concurrent statement. A sufficient condition for safety of a statement a that can be checked locally is that a does not touch any global variables or channels. Indeed, it is this condition that is implemented in Spin.

3 Introducing Discrete Time in Promela and Spin

3.1 Real time Promela

In the discrete time model, time is divided into slices indexed by natural numbers. The actions are then framed into those slices, obtaining in that way a measure for the elapsed time between events belonging to different slices. Within a slice however, we only know the relative ordering between events, as in the time free case. The elapsed time between events is measured in ticks of a global digital clock that is increased by one with every such tick.

In state space enumeration methods like the one used in Spin, verification can be regarded as checking all possible simulations of the system. In that sense the simulation can be considered as a primitive for the validation, so we start with the description of the discrete time implementation for simulation purposes. The basic idea is to execute the system time slice by time slice. Recalling that the execution of statements in Spin is asynchronous and interleaved, the basic problem is how to avoid interleaving of actions belonging to different time slices. One way to solve this is by forcing each process to stop the execution after it has executed all the actions for the current time slice, waiting to receive a signal that the system has passed to a new time slice. This synchronization is done by a special "deamon" process that is not a part of the modelled system and waits in the background to become active only when all the other processes from the system are blocked. This deamon process is a pacemaker that transfers the system to the next time slice, by sending unblocking signals to the other processes.

We implement this synchronization scheme by extending Promela with a new variable type `timer` corresponding to discrete time countdown timers, three new statements `set`, `expire` and `tick` that operate on the new type, and a special timing process `Timers` which is the deamon process that uses `ticks` to decrease the timer values. The implementation can be done entirely on user level, without any additional changes in the Spin source code, for example, with the following Promela macro definitions and timer process:

```
#define timer int
#define set(tmr,val) (tmr=val)
#define expire(tmr) (tmr==0) /*timeout*/
#define tick(tmr) if :: tmr>=0 -> tmr=tmr-1 :: else  fi

proctype Timers()
{ do :: timeout -> atomic{ tick(tmr1); tick(tmr2) } od }
```

The first macro defines `timer` as a synonym for the integer type The `set` macro sets the value of the timer `tmr` to `val`. The `expire` macro is a test which becomes true when `tmr` becomes 0. The `tick` macro is used only in the `Timers` process and it decreases the value of `tmr` provided that it is active, i.e., its value is non-negative. Timers with negative values are considered as deactivated. `Timers` consists of an endless `do` iteration that realizes the passage of time. It is run concurrently with the other processes of the system. The key idea of the concept is the usage of `timeout` - a predefined Promela statement that becomes true when no other statement within the system is executable. By guarding `ticks` with `timeout`, we ensure that no process will proceed with an action from the next time slice until the other processes have executed all actions from the current time slice[1].

Within a process, statements are divided into time slices by putting `set` and `expire` at the beginning and end, respectively, of each time slice. For instance, assuming that we have declared `timer tmr`, and that `A`, `B` and `C` are nonblocking Promela statements, the sequence

```
set(tmr,1); A; B; expire(tmr); C
```

means that `A` and `B` will be executed in same time slice, while `C` belongs to the next time slice. The `expire` statement is a synchronization point where the process waits for `tmr` to become zero. This can be done only by `Timers`, i.e., only when all active timers are decreased. Thus, it is guaranteed that `C` will be executed in the next time slice. In fact, `set` is only a labelling of the time slice (some sort of reading of the global digital clock we assumed in our time model) and it can be permuted with the statements from the time slice that it labels (in our case `A` and `B`).

Also, in cases where we have empty time slices, we optimize sequences of the form

```
set(tmr,1); expire(tmr); set(tmr,1) expire(tmr); ... ; set(tmr,1) expire(tmr);
```

by

```
set(tmr,val); expire(tmr);
```

where `val` is the number of `set`-`expire` pairs.

It is often convenient to use derived macros for modeling various delays. For instance, one tick delay and unbounded nondeterministic delay are implemented, by the following macros

```
#define delay(tmr,val) set(tmr,val); expire(tmr)
#define udelay(tmr) do :: delay(tmr,1) :: break od
```

In the unbounded nondeterministic delay, at each iteration a nondeterministic choice is made whether the loop will be broken and the process will proceed with the execution of a new statement, or the decision will be delayed for the next time slice. In a similar way a nondeterministic bounded delay up to a certain number of ticks, or a nondeterministic delay within lower and upper bounds can be modeled.

It is worth noting that `timeout` was introduced in Promela to make up for the lack of quantitative treatment of timing and it would have been in fact useless in a time setting. Instead, in our concept it is given one of the central roles. Using `timeout` we continue to use the built-in Spin process scheduling and synchronization. In that way we avoid the introduction of additional ”housekeeping” processes and variables. For instance, elements like process scheduler, semaphore variables or some other synchronization mechanisms, might be needed. It is very likely that we will lose in the clarity of the specifications and growth of the state space if all of them are implemented on user level.

[1] There is one `tick` for each timer in the system, In the example above we assumed that there are only two timers, `tmr1` and `tmr2`. The `ticks` are wrapped in `atomic` statement in order to avoid unwanted unblocking of some of the system processes before all timers are decreased.

3.2 Example

In this section we show how the discrete time can be used for the specification and verification of the Parallel Acknowledgment with Retransmission (PAR) protocol [21]. The choice of PAR was motivated by the fact that it is a relatively simple protocol, yet it is complex enough that its correct functioning depends on the duration of time intervals in a nontrivial way. PAR has been used in similar studies as this one, cf. [23, 16].

PAR is one-way (simplex) data-link level protocol intended to be used over unreliable transmission channels which may corrupt or lose data. There are four components in our implementation: a sender, a receiver, data channel K and acknowledgment channel L. The sender receives data from the upper level and sends them labeled with a sequence number that alternates between 0 and 1 over the channel K. After that it waits for an acknowledgment via the channel L. If this does not occur after some period of time, the sender times out and resends the old data. If the data are received undamaged and labelled with the expected sequence number, the receiver delivers them to the upper level and sends an acknowledgment.

Of crucial importance here is the duration of the time-out period which should be longer than the sum of the delays through the channels and the message processing time by the receiver. A premature timeout can cause the loss of a frame by the following scenario: The sender sends a frame and times out too early. This causes a duplicate of the just sent frame to be sent too. The receiver receives both frames and sends two acknowledgments. When the sender gets the acknowledgment for the first frame it thinks that it is for the second one (the duplicate). After receiving this first acknowledgment it sends next frame, which is lost by the data channel. In the meantime the second acknowledgment arrives (which was sent by the receiver as a result of successful receiving of the duplicate) and the sender mistakenly thinks that it is an acknowledgment for the last frame and never resends it.

The complete listing of the discrete time Promela model of PAR is given on the next page. The Promela model starts with the already described preamble of macros for discrete time. Then, after the definition of protocol specific parameters, the timers are declared. There is one timer per protocol entity, i.e., for sender (`sc`), receiver (`rc`), channel K (`xk`) and channel L (`xl`). Although used locally in this model, the timers are declared on global level in order to be accessible for `Timers`. The four defined Promela channels `A`, `B`, `C` and `D` have capacity 0 meaning that they are synchronous rendez-vous channels. `A`, `B`, `C` and `D` are used only as auxiliary channels, because the "real" unreliable channels are modeled as `proctypes`, i.e. as a separate processes. The channels `A` and `B` are the input and output, respectively, to the channel `K`. They deal with messages consisting of two fields — one of type `byte` and one of type `bit`, corresponding to the message contents and the alternating bit. Similarly, `C` and `D` are the input and output for `L` and they can carry messages containing just one bit — the acknowledgment.

After the declarations of local variables, the `Sender` process begins with an unbounded start delay of the first operation which fetches a message from the upper level (user). In our model this is simplified by allowing `Sender` to generate the message itself by the statement `mt = (mt+1)%MAX`. The message followed by a sequence number is sent through the channel `K`.

```
/*discrete time macros*/

#define timer int
#define set(x,y) x=y
#define expire(x) (x==0)
#define tick(x) if :: x>=0->x=x-1; :: else; fi
#define on(x) (x!=-1)


#define delay(x,y) set(x,y); expire(x);

#define udelay(x) do :: delay(x,1) :: break od

/*PAR time parameters*/
#define dK 3 /* delay along channel K */
#define dL 3 /* delay along channel L */
#define dR 1 /* receiver's message processing time */
#define To 9 /* sender's timeout value */
#define MAX 8 /*max number of different message contents*/

/*timers*/
timer sc, rc, xk, xl;

proctype Timers()
{
   do
   :: timeout ->
   atomic{tick(sc); tick(xk);
          tick(rc); tick(xl);}
   od;
}

/*channels*/
chan A = [0] of {byte, bit};
chan B = [0] of {byte, bit};
chan C = [0] of {bit};
chan D = [0] of {bit};

/*protocol entities*/

proctype Sender()
{
  byte mt; /* message data */
  bit  sn=0; /* sequence number*/
  R_h:
    /*message can be sent in any time slice*/
     udelay(sc);
    mt = (mt+1)%MAX;
  S_f:
    A!mt,sn; /*sand through channel K*/
    set(sc,To);
  W_s:
    if
    :: D?_ ->
       if
       :: atomic{skip; delay(sc, 1); sn=1-sn; goto R_h;};
       :: atomic{printf("MSC: ACKerr\n");  goto S_f};
       fi;
    :: expire(sc) -> goto S_f;  /*timeout*/
    fi;

}

proctype Receiver()
{

 byte mr, me=1; /* received and expexted message*/
 bit rsn, esn=0; /*received and expected sequence number*/
 W_f:
   B?mr,rsn;
   if
   :: rsn == esn -> goto S_h; /*correct message and seq. num */
   :: rsn == 1-esn -> goto S_a; /*correct message, wrong  seq. num */
   :: atomic{printf("MSC: MSGerr\n"); goto W_f;};
   fi;
 S_h:
   /*Out!mr*/
   assert(mr == me);
   atomic{delay(rc,dR);
   /*message processing delay*/
   esn = 1-esn; me = (me+1)%MAX};
 S_a:
   C!1; /*send ack through channel L*/
   atomic{delay(rc, 1); goto W_f;};
}
```

```
/*channel processes*/

proctype K()
{
  byte rd; /*received chunk*/
  bit rab; /*received and expected bits*/

  do
  :: A?rd,rab;
     delay(xk, dK);
     if
     :: skip; B!rd,rab;
     :: skip;
     fi;
  od;
}

proctype L()
{
  do
  :: C?_;
     delay(xl,dL);
     if
     :: skip; D!1;
     :: skip;
     fi;
  od;
}

init
{   atomic{set(sc,-1); set(rc,-1);
           set(xk,-1); set(xl,-1);
           run Timers();
           run Sender();
           run K();
           run L();
           run Receiver();}
 }
```

This event is “labeled” by an activation of the timer `sc` which is needed for the usual timeout mechanism (not to be confused with the `timeout` Promela statement) for unblocking the protocol in case of message or acknowledgment loss. After sending the message the `Sender` blocks waiting for the arrival of an acknowledgment or the expiration of the timeout timer. In the first case a nondeterministic choice [2] is made between signalling an error because of a corrupted acknowledgment and accepting the acknowledgment as correct.

`Receiver` starts by waiting for a message. This is achieved by the statement `B?mr,sn`, which denotes the reception of a message through the channel `B`. The receiving statements in our implementation are treated as executable as soon as possible, i.e. in the same slice when the corresponding sender is ready to send. After the label `S_h` an `assert` is used to ensure that the received message always matches the expected one. (Note that this statement is added to the model only because of validation reasons.) The message processing latency is modeled by a `delay` on the `Receiver`'s timer. Also the expected message and sequence number are updated. Successful reception is confirmed by sending a message through `L` (via Promela channel `C`) and after a latency delay the `Receiver` starts waiting again to a new message.

Channel processes model the lossy channels `K` and `L`. Both processes have the same structure — after a delay they nondeterministically chose between delivering or losing the message. The `skip` is always executable. It is used here to give equal precedence for both the choices.

All processes are started from a special process called `init` via `run` statements. By enclosing the `run` statements within an `atomic` it is ensured that they start simultaneously.

Verification of the model can be done like in the standard Spin. Among the other properties, the validator is able to find out and display the scenario of losing a message, in case of an erroneous combination of the timing parameters.

3.3 Low level implementation

In order to achieve more efficiency we get down from the high-level implementation to a lower level, by realizing the `tick` construct in the Spin's source code. Also, the new semantics of `tick` is adopted so as to decrease all the timers in the system together. The implementation of the new statement in Spin is completely modular and does not interfere with other features.

The timing procedure now becomes much simpler:

```
proctype Timers
{ do :: timeout -> tick od }
```

As it is universal for all models, it can be made part of the standard header file that contains the basic macros. Another advantage of the new `tick` is that the timers can be declared locally. Like ordinary local variables, local timers can be dynamically created and deleted together with the processes they belong to. To the best of our knowledge this possibility is not present in any tool for validation of timed systems. Variable number of timers automatically means saving of the state space, due to Spin's variable size state representation.

[2] The `if` statement is similar by its semantics to the `do` statement - a random choice is made between the alternatives and if none of them is enabled than the statement is blocked

The most important benefit of the local usage of timers is the partial order reduction which is, as already emphasized, one of the main strengths of the enumerative methods. The integration of partial order reduction with the discrete time model is one of the main results of this paper and it is the topic of the next subsection.

3.4 Compatability with Partial Order Reduction

In this section we prove that the partial order reduction algorithm of (standard, untimed) Spin is correct for the timed extension as well. This algorithm is based on selecting, during the DFS exploration of the state space, only statements from the ample set. The construction of an ample set is based on the notion of *safety* of statements: given a criterion to determine which statements are safe, the underlying theory, that was presented in Section 2.1, guarantees that the reduction strategy is correct, i.e. that (LTL) properties are preserved. For untimed Spin, this criterion is that a statement does not read or write any global variables or channels. We will now show that this criterion is still sufficient in the presence of timed statements.

Lemma 3.4.1 *Any statement that is safe in standard, untimed Promela, remains safe in the presence of the new timed statements* `set`, `expire`, *and* `tick`. *Furthermore, when applied to local*[3] *timers, the statements* `set` *and* `expire` *are safe, and* `tick` *is independent from any other statement.*

PROOF In order to show that a statement is safe, we have to show that it is *invisible*, and *independent* from any concurrent statement (i.e. any statement with a different Pid).

Let a be a statement in untimed Promela that is safe. Then it is clearly still invisible in timed Promela. Also, a is obviously still independent from any concurrent untimed statement. a's independence from any ("new") timed statement that is concurrent follows from the fact that the set of timer variables, that are the only variables which can be read or written by timed statements, is disjoint from the set of ordinary variables which can be affected by a.

Next, we let a be one of the statements `set` and `expire`, applied to a local timer. a is invisible because the local timer variable may not occur in the property to be checked. Let b be a concurrent statement different from `tick`. Then a is obviously independent from b because of the disjoint sets of variables they operate on. Independence between `tick` and a, and indeed independence between `tick` and any timed or untimed statement, follows from the fact that `tick` can never be enabled simultaneously with another statement. This is because of the way `tick` is implemented: it is only enabled in the case of a `timeout` condition.

As `tick` is implemented in the Spin source code, is is not subject to reduction by the partial order algorithm. However, this does not have any repercussions: any `tick` statement, regardless whether it is acting on a local or on a global timer, is the only statement that is enabled, when it is enabled at all. So, no reduction would have been possible anyway.

[3] Recall that a local timer is a timer variable that is declared local to a Promela process. In timed Spin, it can be manipulated by the declaring process or by the `tick` operation.

4 Experimental Results

We have tested the implementation on various models known in the literature (e.g. Train Gate Controller, Seitz Circuit, Leader Election Protocol). We focus our attention on three of them that illustrate the effectiveness of the implementation and particularly the partial order reduction: Fischer's mutual exclusion protocol, the already presented PAR protocol and the Bounded Retransmission Protocol (BRP). All tests were performed on a Sun SPARC Station 5 with 64 MBytes of memory.Besides partial order reduction we used as an additional option *minimized automata*, a technique for reduction of the state space recently included in the standard Spin distribution. In the *options* column of the tables below, "n", "r" and "ma" denote verifications without POR, with POR, and with POR together with minimized automata, respectively.

The version of the Fischer's protocol that was verified is a translation of the same model written in Promela with real (dense) time of [22], with virtually the same timing parameters. The obtained results for the verification of the mutual exclusion property are shown in Table 1 (N is the number of processes). As expected, the state space growth is exponential and we were able

N	*option*	*states*	*transitions*	*memory* [MB]	*time* [s]
2	n	528	876	1.453	0.1
	r	378	490	1.453	0.1
	ma	378	490	0.342	0.2
3	n	8425	10536	1.761	0.8
	r	3813	4951	1.596	0.4
	nr	3813	4951	0.445	0.3
4	n	128286	373968	7.085	15.5
	r	34157	44406	3.132	2.8
	ma	34157	44406	0.650	33.7
5	r	288313	377032	17.570	36.2
	ma	288313	377032	1.059	332.5
6	ma	$2.35E^6$	$3.09E^6$	2.118	6246.1

time parameters				*option*	*states*	*transitions*	*memory* [MB]	*time* [s]
dK	dL	dR	To					
3	3	1	9	n	1318	1822	1.453	0.2
				r	1116	1533	1.493	0.3
30	30	10	90	n	7447	9994	1.761	0.5
				r	7295	9705	1.801	0.6
300	300	100	900	n	68737	91714	5.135	4.9
				r	68585	91425	5.420	4.9
				ma	68585	91425	2.221	143.7

Table 1: Results of Fischer and PAR.

to validate the model without using POR up to 4 processes. For 6 processes even POR was not enough and it had to be strengthened with minimized automata[4]. Nevertheless, the profit from POR is obvious and even becomes more evident as the number of processes grows. While for $N = 2$ the number of states is reduced to 72% compared to the case without POR, and the transitions are reduced to 56%, for $N = 4$ the reduction increases to 27% and 12% for states and transitions, respectively.

It is difficult to compare our implementation with the one from [22], because they are based on different time models. Nevertheless, having in mind that the property that was checked was a qualitative one, for which discrete time suffices [11], one can safely say that after $N = 4$ our implementation has better performance. In fact, for $N = 5$ the validator from [22] runs out of memory. Obviously, POR is the decisive factor, because without POR our implementation is also incapable to handle cases for $N > 4$.

In the case of the PAR protocol the reduction is very small and even diminishes with the increase of parameters (Table 1). The insensitivity of PAR model to reduction can be explained by observing that PAR is in fact a sequential algorithm. The protocol entities take turns during the

[4]Unfortunately, although the automata minimization is very effective in the reduction of memory consumption, it seems that this improvement is achieved on the expense of an exponential growth of the verification time.

option	*states*	*transitions*	*memory* [MB]	*time* [s]
n	32967	71444	4.116	6.7
r	12742	14105	2.517	3.2
ma	12742	14105	1.571	26.4

option	*states*	*transitions*	*memory* [MB]	*time* [s]
n	199454	658852	7.515	37.6
r	113566	294546	5.547	19.0

Table 2: Results for timed and untimed BRP.

execution of the protocol, most of the time only one of them being active while the others are waiting passively for some trigger-event. Very little concurrent activity results in a bad behaviour of the POR algorithm which deals with concurrent actions from different processes. It is an opposite situation compared to Fischer's protocol where the higher degree of concurrency led to an improvement of the effects of POR.

Note the sensitivity of the state space to the increase of parameter values. This sensitivity to parameters is analogous to the same effect observed in the region automaton implementation of the dense time system based on timed automata [1, 2]. At least for the type of protocols like PAR, when the sequential component of the system is predominant, this problem can be solved by grouping together consecutive ticks into one tick to avoid generation of spurious intermediate states.

Our implementation of BRP is based on [6], where the protocol was treated using a combination of Uppaal (for timing aspects) and Spin (for consistency of the specification). We give the results for safety properties (absence of deadlock and assertions). Using the same parameters as in [6], all the properties verified there can be checked with a memory consumption of at most 5 Mbytes and less than a minute of CPU time. This is a much better result than the one in [6]. In addition, our model has the advantage that the validation is done completely within Spin.

The benefit of POR reduction is obvious — the number of states is reduced to 39%, while the transitions are reduced to 20%. In order to get an indication how well the partial order reduction combines with our time extension in Spin, we have compared the above reductions for the BRP to the partial order reduction that is achieved on a slightly different version of the protocol that does not involve timers (see e.g. [8] or [7]). The result is given in Table 2. The effect for the timed version turns out to be even better: 39% and 20% versus 57% and 45%. It has to be said though that in order to model the effect of timeouts in the untimed case, we need an additional global variable, which may influence the effects of the reduction. Anyway, this is a promising result having in mind that, unlike previous two examples which are more academic, BRP is just slightly simplified version of an industry protocol used by Philips, which gives us hope that POR could work for real-world applications.

5 Towards Dense Time

The same idea of using a special deamon process to manage time passage can be used to extend Promela and Spin with dense time. In the sequel we assume that the reader is familiar with the theory of timed automata (see for instance [1]), which we will use as a basic model for the timing extensions. More precisely, we will consider each Promela process in the system as a timed Büchi automaton. The system as a whole is then represented by a timed automaton obtained as a standard composition (c.f. [1]) of the automata corresponding to the processes. We will show how the region automaton that corresponds to this timed automaton can be implemented.

Let us suppose that, in order to represent timing, we have extended the syntax of Promela like in [22] with a new variable type `clock` to represent dense time clocks and timed statements of the form

`when {guard} reset {clocks} untimed_statement`

The timed *guard* of the statement is a conjunction of inequalities of the form $x \sim c$, where x is a clock variable, c is a nonnegative integer, and $\sim \in \{<, \leq, =, \geq, >\}$; *clocks* is a list of clocks that are reset by the statement; and *untimed_statement* is a ordinary Promela statement. These timed statements are actually the edge transitions of the underlying timed automata.

The key observation is that, because of the particular form of the guard predicates, to implement time passage in the system it is enough to use a *discrete* global clock which *ranges over clock regions*, which can also be represented as conjunctions of inequalities over clock variables. One can check the guard conditions by checking whether the clock region which is the current value of our global clock satisfies the guard predicate. The number of regions is always finite [1] which guarantees termination of the validation procedure. Once having a discrete clock, the whole concept of a special procedure that implements the time passage can be applied like for the discrete time.

Thus, the value of the global clock is changed by the process `Clocks` via `tick` statements that compute the successor regions. The global clock value can also be changed by timed statements that reset the clocks listed in their `reset` parts. Like in the discrete time model, `tick` has the lowest priority among the statements. This can cause an anomaly in our implementation in that the timed statements are executed as soon as they are enabled, i.e. only in the first clock region that satisfies the guard. To avoid this, analogously to the unbounded start delay construct in the discrete time model, we ensure that every action that is enabled both in the current clock region and in its successor region can nondeterministically "choose" whether it will be executed in the current region or be postponed to the next clock tick[5]. This can be implemented by wrapping each timed statement into a nondeterministic choice according to the following basic scheme given in "quasi-Promela":

```
back:
if
:: when{guard} reset{clocks} untimed_statement
:: /* shadow statement */ delay_one_tick(guard); goto back
fi
```

The choice offers an alternative "shadow" delay statement which waits for the new clock tick, provided that the guard of the original statement is satisfied also for the successor region. After that the control jumps back at the beginning of the `if` statement.

It is noteworthy that in the discrete time case the clock managing process also ticked the time equivalence classes, which for this model were vectors with integer valued components.

In the proposed implementation we do not need invariants to enforce urgency of actions. Instead, the conditions on clocks of the discrete time transitions, (i.e. timed guards of the state-

[5]Of course, in the meantime (between the two ticks) the system can jump via discrete transition to a new region when the statment is no longer enabled.

ments) play a role of *deadlines*, similarly like in [20]. The time transition must not happen, if it can disable all statements for some process.

In an analogous way like for the discrete time case, the untimed POR algorithm can be adapted for region automata. The proof for this is similar to the proof given for the discrete time. It is enough to see that the operations on local clocks are safe. An additional observation that is needed is that if the "shadow" one-tick-delay statement has as a guard an expression on only local clocks, it is independent from all the other statements not belonging to the same process it belongs to. For the clock tick holds the same remark about "well fitting" into POR, as in the discrete time case.

6 Conclusions and Future Work

We presented an extension of Promela and Spin with discrete time, using integer variables as timers to stamp the time slices. The core idea was to use a special background deamon process to implement the passage of time. For this purpose we used Promela `timeout` predefined statement as a mechanism for process (i.e. timer) synchronization. We first showed how the concepts can be implemented using only the existing features of Promela and Spin, by extending the language with new time statements as Promela macro definitions. The usage of timing features was demonstrated on the specification and validation of PAR protocol.

As one step further toward a more efficient integration of the timing framework into Spin, we implemented the timing on the level of the validator's source code. With this low level implementation, we gain the possibility to use the timers locally. This provides dynamic change of the number of timers in the system - a feature which to the best of the authors' knowledge does not exist in any implementation of verification of real-time systems. This is in accord with the possibility of dynamic creation of processes already present in Spin. But, the most important benefit of the local usage of timers, and one of the main results of this paper, was the adaptation of the existing partial order reduction algorithm to the discrete time setting. The urgency that is a necessary feature in modeling of a broad class of interesting timing dependent systems is achieved in a natural way, without usage of invariants. The low level implementation is seamless and completely compatible extension of the standard Spin validator which allows one to use all the existing features of the untimed validator in a standard way. We showed the efficiency of the partial order and the implementation in general on several examples, of which we emphasize the verification of the Bounded Retransmission Protocol, as an industrial protocol used by Philips. The main future tasks will certainly be to test the approach and accumulate experience by doing new verifications on systems known in the literature or some new real-world (industry) cases which are time dependent.

Besides the improvement of the existing prototype, the main direction for the future work will be the implementation of dense time in Promela and Spin, along the same concepts that were used for discrete time and following the guidelines described in the previous section. This will provide a possibility to express and verify more general class of properties.

It would be very interesting to incorporate the ideas about data and time abstraction of [5] both in the existing discrete time prototype, as well as in the future dense time implementation, in order to obtain verifications that are independent of the concrete parameter values, and to see how general is their applicability.

Another promising idea is to extend the concept of time managing process to a class of hybrid systems called discrete time rectangular automata [9]. This can be done by introducing, besides timers(clocks), new time dependent variables and allowing the timing process to change their value also with steps greater than one.

Acknowledgments: The first author would like to thank for the help during the work on this paper Stavros Tripakis and Bart Knaack.

References

[1] Alur, R., Dill, D.L., *A Theory of Timed Automata*, Theoretical Computer Science, 126, pp.183-235, 1994.

[2] Alur, R., Kurshan, R.P., *Timing Analysis in Cospan*, Hybrid Systems III, LNCS 1066, pp.220-231, Springer, 1996.

[3] Baeten, J. C. M., Bergstra, J. A., *Discrete Time Process Algebra*, Formal Aspects of Computing 8 (2), pp. 142-148, 1991.

[4] Bosnacki, D., *Implementing Discrete Time in Promela and Spin*, International Conference on Logic in Computer Science, LIRA '97, University of Novi Sad, Yugoslavia, 1997.
Copy also available from the author.

[5] Dams, D., Gerth, R., *Bounded Retransmission Protocol Revisited*, Verification of Infinite Systems Infinity, Bologna, Italy, 1997.

[6] D'Argenio, P. R., Katoen, J.-P., Ruys, T., Tretmans, J., *The Bounded Retransmission Protocol Must Be on Time!*, TACAS'97, 1997.

[7] Graf, S., Saidi, H., *Construction of Abstract State Graphs with PVS*, Computer Aided Verification '97, LCNS 1254, pp. 72-83, Springer, 1997.

[8] Groote, J. F., van de Pol, J., *A Bounded Retransmission Protocol for Large Data Packets*, in Wirsing, M., Nivat, M., ed., Algebraic Methodology and Software Technology, LCNS 1101, pp. 536-550, Springer-Verlag, 1996.

[9] Henzinger, T. A., Kopke, P. W., *Discrete-Time Control for Rectangular Automata*, International Colloquium on Automata, Languages and Programming (ICALP 1997), LNCS 1256, pp. 582-593, Springer-Verlag, 1997.

[10] Henzinger, T. A., Ho P.-H., Wong-Toi, H., *A Model Checker for Hybrid Systems*, Computer Aided Verification 97, LNCS 1254, pp.460-463, Springer-Verlag,1992.

[11] Henzinger, T. A., Manna, Z., Pnueli, A., *What good are digital clocks?*, Proceedings of the ICALP'92, LNCS 623, pp.545-558, Springer-Verlag, 1992.

[12] Holzmann, G. J., *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`

[13] Holzmann, G., Peled, D., *An Improvement in Formal Verification*, FORTE 1994, Bern, Switzerland, 1994.

[14] Larsen, K. G., Pettersson, P., Yi, W., *UPPAAL: Status & Developments*, Computer Aided Verification CAV 97, LNCS 1254, pp.456-459, Springer-Verlag, 1992.

[15] Kesten, Y., Manna, Z., Pnueli, A., *Verifying Clocked Transition Systems*, Hand-outs of School on Embedded Systems, Veldhoven, The Netherlands, 1996.

[16] Klusener, A. S., *Models and Axioms for a Fragment of Real Time Process Algebra*, Ph.D. Thesis, Eindhoven University of Technology, 1993.

[17] Knaack, B., *Real-time extension of SPIN - Ongoing Research*, Abstracts from SPIN'97, Third SPIN Workshop, Twente University, Enschede, The Netherlands, Also: `http://netlib.bell-labs.com/netlib/spin/ws97/papers.html`

[18] Pagani, F., *Partial Ordrers and Verification of Real Time Systems*, Formal Techniques in Real Time and Fault Tolerant Systems FTRTFT 96, LNCS, Springer, 1996.

[19] Peled, D., *Combining Partial Order Reductions with On-the-Fly Model Checking*, Computer Aided Verification 1994, LCNS 818, pp. 377-390, 1994.

[20] Sifakis, J., Yovine, S., *Compositional Specification of Timed Systems (Extended Abstract)*, STACS 1996.

[21] Tanenbaum, A., *Computer Networks*, Prentice Hall, 1989.

[22] Tripakis, S., Courcoubetis, C., *Extending Promela and Spin for Real Time*, TACAS '96, LCNS 1055, Springer Verlag, 1996.

[23] Vaandrager, F., *Two Simple Communication Protocols*, in Baeten, J.C.M., ed., Applications of Process Algebra, pp.23-44,Cambridge University Press, 1990.