Extending Promela and Spin with Discrete Time

Dragan Bošnački

Department of Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 Eindhven, The Netherlands
e-mail: dragan@win.tue.nl
fax: +31 40 246 3992

Abstract. An implementation of discrete time in Promela, a high level modeling language for specification of concurrent systems, is presented. Discrete time features are introduced on user level, using only the existing language constructs and without any change in Spin, its associated software package for automated analysis and validation. It is shown how the new time framework can be used for simulation and validation of systems which correct functioning depends crucially on timing.

1 Introduction

Promela is a high level modeling language for specification of concurrent systems. The models (programs) written in Promela are used as an input for *Spin* software package for their automated simulation and validation.

The time ordering of the actions in Promela is implicit and is given by the sequential composition of statements, like in the ordinary procedural programming languages. The time relations are only *qualitative*, meaning that we only know that an action will eventually be executed after another, but we do not know the exact time interval that will elapse between the two events.

This can be a significant shortcoming when the systems which correct functioning closely depend on the timing parameters are simulated or verified. For instance, many such examples can be found among communication protocols that have to deal with unreliable transport media, where the duration of time-out intervals could be of utmost importance.

To overcome this problem of timing the formal verification methods are extended with the concept of *continuous* or *discrete* time. Usually this is done by labeling each action with a number representing the time moment when it is happening. The *continuous (dense) time* is modeled by real numbers, while the *discrete time* is represented using integers.

To the best of author's knowledge, the only completed attempt to extend Promela and Spin with time is [19], and there are also the ongoing researches of [17] of [7].

In [19] the dense time model is used and the implementation of timing is done appropriate changes in the sources of the Spin package. However, the implementation is based on the old model of timed automata without invariants, which lacks urgency. Consequently, it is impossible, or at least very difficult and ineffective, to model a broad class of interesting systems, including common types of communication protocols, for instance.

The research of [17] to the best of our knowledge is trying to cope with this problem, but the results are still unclear.

Here we adopt a less ambitious approach by taking discrete time and upgrading Promela (and indirectly Spin) with timing *entirely on user level*, i.e. using only the existing capabilities of the language, without any intervention in the software package.

The work of [7] is similar in this regard, because it is based on the models of synchronous discrete time automata, written on user level.

Our choice of discrete time and high-level implementation can be justified by several reasons.

The discrete time is easier to treat then the real time and on the other hand, it can be satisfactory and more effective for a broad class of practical applications [12]. Under certain restrictions the validation results obtained under the integer time model, are fully valid for the dense time model. This is the case for all validations when the system requirements are only qualitative, i.e., do not contain explicit timing relations. Other examples are bounded response and bounded invariance properties, which cover a large class of applications.

Further, time free Promela and Spin have been in the arena of model checking for years and it is natural to reuse the existing tools taking advantage of the experience that has been gained. This can be achieved only by a seamless solution which should be modular and independent as much as possible of the other parts of the Spin code.

Although it can be expected that it would be less effective than the implementation of timing features on lower source-code level, the high level approach can be justified by its flexibility. If we have to extend Spin with any new feature - like probabilities, for instance - as long as the implementation of this new feature does not change the syntax and semantics of the statements that are used for discrete time implementation, we can automatically have discrete time added to the new concept.

Another significant argument which justifies the interest in discrete time is the recent technique of [7] of data abstraction and especially of timing abstraction for discrete time, which provides verification of systems that is virtually independent of the concrete values of the data and time parameters. It is still to be seen how it can be transferred to real time.

In the proposed implementation the urgency is achieved in a natural manner such that problems like in [19] are no longer present.

It was also a challenge to show that the new features, introduced in the recent versions of Promela and Spin, provide enough power to express the time characteristics of the parallel systems.

Finally, the high-level implementation can be regarded as a first step toward a low-level implementation by adding the new timing features directly in the Spin source code. The first versions of this kind of low-level implementation already realized and the first results are very positive [6].

Paper layout: The rest of the paper is structured as follows. The next section is a brief description of Promela, Spin and Xspin. Section 3 is the core of the paper where the basic ideas for implementing discrete time in Promela are presented. The application of the new timing features for system specification and verification is shown in Section 4, on the example of PAR protocol. The last section concludes the paper with some considerations concerning the future work.

2 Spin, Xspin and Promela

Originally, Promela and Spin have been developed for analysis and validation of communication protocols. The language syntax is derived form 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 being beyond the scope of this paper, we assume that the reader is familiar with the basics of the language and the validator. Following [14] we only give a short overview of the language and validation tools. We suggest [13] as a further reference.

In Promela, the 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 actions is considered asynchronous and interleaved, which means that in every step only one *enabled* action is performed and without any additional assumptions of the relative speed of the process execution.

Given as an input a Promela model Spin can do random or interactive simulations or to generate a C program that performs a validation of the system by scanning the state space.

The validation with the compiled C program is done in separate phases. The one phase is validation of the *safety* properties (absence of deadlock, unspecified message receptions, invalid end states and assertions) through a search of the state space. Verification of most of the *liveness* properties can be handled by detection of progress and acceptance cycles. The never-claims constructs are the most general way to verify the system by providing the possibility to express properties as linear-time temporal formulae. All kinds of errors are reported by saving the trace of the actions leading to an invalid state or cycle. The erroneous sequence can be repeated as a guided simulation.

The simplest way to do scanning of the state space is through an exhaustive search. For large state spaces some more sophisticated methods are used as state-vector compressing, partial-order reduction and bit-state hashing. The last technique only gives approximate results in a sense that there is only a guarantee that some part of the state space is covered.

Xspin is a graphical interface for Spin.It provides an integrating windowing environment for writing Promela models and carrying out virtually all Spin functions in a user friendly manner. The outputs are displayed in various ways, among which are Message Sequence Charts.

3 Implementation of Discrete Time in Promela

In the discrete time model used in the sequel time is divided into equal slices indexed by natural numbers. The actions are then framed into those slices, obtaining in that way a good *quantitative* estimation for the intervals between the events belonging to different slices. Within a slice however, we can only have the qualitative relation between the events, as in the time free case.

The time elapsed between the events is measured in ticks of a global digital clock which is increased (decreased) by one with every such a tick. Obviously the actions between two consecutive ticks have the same integer time stamp and only a qualitative time ordering between them can be established. (The time stamps can be regarded as obtained by a truncating of the fractional part of the time measured by a real valued global clock(s)).

The model used in this paper coincides with the one from [3], or with the *fictitious clocks* time model from [1].

In the state space enumeration methods like the one used in Spin, the validation can be regarded as execution and checking of all possible simulations of the system. In that sense the simulation can be considered as a primitive for validation, so we start with the description of the discrete time implementation for simulation purposes.

This can be done in a straightforward way by introducing one global integer variable and a special process as in the following segment

```
int TMaster;

proctype MasterClock()
{
  do :: timeout->TMaster=TMaster+1; od
}
```

The variable TMaster is acting as a master clock on which all the processes are then synchronized, by forcing them to wait for the next clock ticks to continue their execution.

For instance, we can implement a delay of one tick with the following sequence:

```
T=TMaster; TMaster==T+1 -> other_statements
```

where `T` is a local integer variable. A process containing this sequence will hang on the boolean condition until `TMaster` is incremented by one. In that way the execution of `other_statements` will be postponed for the next time slice.

The value of `TMaster` can be increased only by the process `MasterClock`, which consists of an endless `do` loop. An instantiation of `MasterClock`, started form `init` process, 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. Its usage is allowed only in `Clocks` process.

By guarding the increase of the clock with `timeout` we ensure that no process will proceed with an action from the next time slice until all the other processes have executed all actions from the current time slice. (Within the same time slice the actions can be interleaved in an arbitrary way, of course.) Putting `timeout` at the beginning of each iteration causes `MasterClock` to have the least priority of all processes. Thus, it has to wait for all of them to finish the execution of their actions for the current slice, and then it can increment `TMaster`. This will trigger the hanging processes and in fact cause the system to pass into a new time slice.

It is worth noting that `timeout` was introduced in Promela to make up 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.

When using Xspin, one can add an appropriate `printf` statement in `Master Clock` so that the changes of `TMaster` can be displayed on the message sequence charts output.

Although it can be improved in several ways, the described straightforward approach would still not be adequate for validation of the discrete time models. Even if we solve the problem of clock overflows, for instance, the virtually unbounded increasing of the master-clock will inevitably lead to a state space explosion.

The remedy is to use multiple clocks instead of just one master-clock. With a careful managing of their operation we will be able to eliminate a lot of unnecessary transitions and shrink the state space significantly. This new approach is given by the following suite of Promela macro definitions:

```
#define clock short
#define OFF (-1)
#define TOVAL (0)
#define set(x,y) x=y
#define shut(x) set(x,OFF)
#define on(x) (x!=OFF)
#define tick(x) if :: on(x)->x=x-1; :: else; fi
#define expire(x) (x==TOVAL) /*timeout*/
#define delay(x,y) set(x,y); expire(x);
```

to which we add the process `Clocks`, which is a modification of `MasterClock` adapted to the new concepts. If, for example, we assume only two clocks in the model, `sc` and `rc`, it could look like on Figure 1.

The first macro defines `clock` as a synonym for the `short` type. We could have as well used any other integer type - the choice of `short` is motivated merely by practical reasons. The special constants OFF and TOVAL determine two

```
proctype Clocks()
{
          do
     :: timeout -> atomic{tick(sc); tick(rc);}
   if
   :: !on(sc) && !on(rc) -> break;
   :: else;
   fi;
  od;
}
```

Fig. 1. Process `Clocks`

special states of the clock - the inactive and timeout state, respectively, that will be described in a moment. The next five definitions that follow reflect the work of clocks as countdown timers. All clocks are shut off in the beginning of `init` process by initializing them to OFF using `shut` . `shut` is only a variant of `set` through which a clock can be set to an arbitrary value. If the clock has been shut off, setting it to a positive value will switch it `on`. Although all the clocks can be set to by any process, they can be decreased only by execution of a `tick` in the `Clocks`. Moreover, a clock value can be decreased only if it is not OFF, i.e. only if the clock is active. The `expire` macro indicates a timeout, a moment of timer expiration when it reaches the timeout value. `delay` can be regarded as a "higher level" macro that uses `set` and `expire` and is used to postpone the execution of a statement for a fixed number of time slices.

It should be said here that the idea for using decreasing clocks was inspired by the work of [7]. Although, it is possible to use the increasing clocks too, which was our first choice, or even to use simultaneously both increasing and decreasing ones, there are at least two advantages of the countdown.

First, the writing of models is more comfortable because one does not need to worry about stopping the clocks with `shut`, when they are not needed any more. This situation can occur after a timeout, for example. This can be a problem because a "forgotten" increasing clock can easily pump up the state space. The countdown clocks are much better in this regard because they are self-deactivating: after a timeout (represented in the concrete model with TOVAL=0) with the next clock tick they will pass into an inactive state (represented by OFF=-1).

Also, we do not need to worry about the possible overflow detection, which otherwise should have been solved by permanent inspection of the clock values in `Clocks`.

There are some new features in `Clocks` that were not present in `MasterClock`. `Clocks` is usually incrementing multiple clocks, instead of one and the clocks are incremented only if they are not shut off. In case all clocks are switched off then the loop is broken and this leads to a deadlock of the system. This feature was necessary in the new time settings, because having `timeout` at the beginning of the endless loop we loose the possibility to detect a deadlock. Now there is a process that is always active - namely the `Clocks` itself! And this is of course artificial, because we have introduced `Clocks` as a mere modeling convenience and it is not a component of the original system. Allowing for loop to be broken, we restore the possibility of deadlock detection.

The usage of `atomic` sequence is to avoid the standard *test and set problem*, i.e. the interference of other processes which otherwise could have been activated by the increase of some of the clocks. Such an unwanted awakening of processes between two ticks in the `Clocks` process could lead to inconsistencies in the synchronization.

The atomic sequences have in general a key role in the reduction of the complexity of validation models. We can even spare some state space by putting the whole do loop of `Clocks` in an atomic clause. (This is possible in the new releases of Spin because the usage of blocking actions inside atomic is legitimate after the version 2.0.) Unfortunately, in that case we will have to limit our verification only on safety properties and acceptance cycles, because the corresponding algorithms used by Spin for non-progress cycles and never-claims, would no longer be compatible with our approach.

The presented starting set of macro definitions is of quite reasonable size. Of course, it can always be extended with new macros to improve the readability and ease the writing of models. For instance, we can introduce the `udelay` unbounded delay for an arbitrary time slices. Such a construct will be necessary if we bare on mind that all the Promela statements in the presented framework are *current time slice* actions, using the terminology of [3]. In addition, we are going to need *any time slice actions* which are not time dependent and can happen in any time slice.

It is interesting to note that by activating and deactivating the clocks we actually implement a *discrete time with relative timing*. It could be useful to draw a parallel between specifications in other formalisms dealing with relative timing, like aforementioned mentioned ACP with discrete relative timing of [3], for instance, and their counterparts in Promela.

4 Validation in Promela with Discrete Time

In this section we show how discrete time can be used for specification and validation of systems in Promela and Spin on the example of Parallel Acknowledgment with Retransmission (PAR) protocol [18].

The choice of PAR was motivated by the fact that it is relatively simple protocol for which a complete model can be given in the paper, but it is yet complex enough that its correct functioning depends on the duration of time intervals in a nontrivial way. Another advantage of PAR is that it occurs often in the literature on specification and verification (e.g. [20, 16]), so it is well-known and also the results obtained with discrete time Promela can be checked against the existing ones.

4.1 Informal Description of PAR Protocol

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 transmission channel K and acknowledgment transmission channel L (Figure 2).

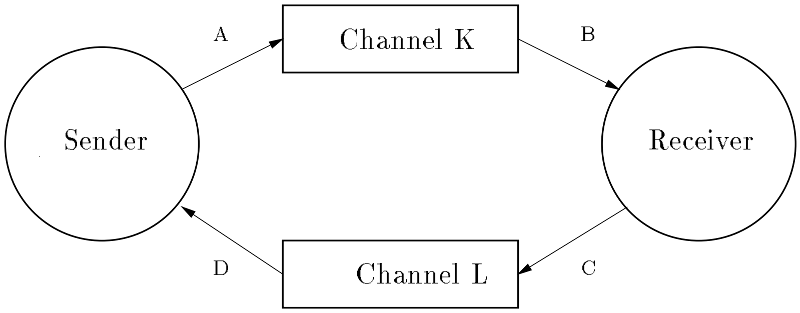

Fig. 2. PAR protocol entities.

The sender receives data from the upper level and sends them labeled with a sequence number over the unreliable channel K. The sequence alternates between 0 and 1, and sometimes PAR is classified as a variant of Alternating Bit Protocol. After that it waits for an acknowledgment, which should be received via channel L, from the receiver before a new datum is transmitted. If an acknowledgment does not occur after some period of time, the sender times out and resends the old data. The receiver receives data from the channel K and if the data are undamaged and labeled with the expected sequence number it delivers them to the upper level and sends an acknowledgment through the channel L to the receiver.

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 message processing time by the receiver. The 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 receives the acknowledgment for the first frame he 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. But 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. In that way this frame never arrives to the final user (upper level).

4.2 A Model of PAR Protocol in Promela with Discrete Time

We give the complete listing and short comment on the discrete time Promela model of PAR protocol.

```
/*discrete time macros*/

#define clock short
#define OFF (-1)
#define TOVAL (0)
#define set(x,y) x=y
#define shut(x) set(x,OFF)
#define on(x) (x!=OFF)
#define tick(x) if :: on(x)->x=x-1; :: else; fi
#define expire(x) (x==TOVAL) /*timeout*/
#define delay(x,y) set(x,y); expire(x);

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

#define shutoff(x,y,z,w) !(on(x)) && !(on(y)) && !(on(z)) && !(on(w))


/*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*/

/*clocking*/

clock sc, rc, xk, xl;

proctype Clocks()
{
          do
             :: timeout -> atomic{
  tick(sc); tick(xk);
                                  tick(rc); tick(xl);
                                  if
                                  :: shutoff(sc,rc,xk,xl) -> break;
                                  /*potential system deadlock*/
                                  :: else
                                  fi;
                                  }
```

```
          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: /*unbounded start delay - sending of a new message
        can start in any time slice*/
        udelay(sc);
        mt = (mt+1)%MAX;

  S_f: A!mt,sn; /*sand through channel K*/
        set(sc,To);

  W_s: do
:: D?_ ->
           if
           :: atomic{skip; delay(sc, 1); sn=1-sn; goto R_h;}; /*ack is OK*/
           :: atomic{printf("MSC: ACKerr\n");  goto S_f};
           fi;
        :: expire(sc) -> goto S_f;  /*timeout*/
        od;

}

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{shut(sc); shut(rc); shut(xk); shut(xl);
          run Clocks();
          run Sender();
          run K();
          run L();
          run Receiver();          } }
```

The Promela model starts with the already described preamble of macros for discrete time, extended with two new macros. The first one is `udelay` that implements an arbitrary unbounded delay of actions. (It is analogous to the unbounded start delay operator from [3].) In each iteration of the `do` [1] statement a nondeterministic choice is made between breaking the loop or continuing with an additional one tick delay.

The second macro `shutoff` is just a short way to write the condition that all the clocks are deactivated and which is used in `Clocks` to break the loop for possible deadlock detection.

After the definition of protocol specific parameters, the clocks are declared. There is one clock per protocol entity, i.e., for server (`sc`), receiver (`rc`), channel K (`xk`) and channel L (`xl`). Although used locally in this model, the clocks are declared on global level in order to be accessible for `Clocks`.

`Clocks` process differs from the one described earlier only by two new `ticks` for the channels' clocks.

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 by `proctype`, i.e. as a separate processes, as shown below. Channels `A` and `B` are an 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 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 the `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` (i.e. its entry - Promela channel `A`). (Note the CSP style syntax of the communication statements.) This event is "labeled" by an activation of the clock `sc` which is needed for the timeout mechanism (should not be confused with the `timeout` Promela statement). After sending the message the `Sender` loops waiting for an acknowledgment. The `do` loop can be broken also by expiration of the timeout clock. When an acknowledgment is received (via `D`) then a nondeterministic choice [2] is made between signaling an error (here represented as an output on the Message Sequence Charts diagram) because of a corrupted message, or accepting the acknowledgment as correct.

[1] The semantics of `do` statement is the following: In each iteration one of the enabled alternatives (statements) is executed. If no guard (statement) is enabled, then the whole statement is blocked, until one of the guards become executable. The iteration can be terminated by explicitly breaking it by `break` or jumping out of it by `goto`

[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

`Receiver` starts by waiting for a message. This is achieved by the statement `B?mr,sn`, which denotes the reception of message through the channel `B`. The Receiver is blocked until a message arrives. The receiving statements in our implementation are treated as executable as soon as possible. /* if statement */

After the label `S_h` the assertion that the received message is equal to the expected one is checked. The message processing latency is modeled by a `delay` on the `Receiver`'s clock. Also the expected message and sequence number are updated.

The 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` statement is a synonym for 1 and it 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` statement. By enclosing the `run` statements within an `atomic` it is ensured that they can start simultaneously.

Given as an input to the Spin validator (version 2.9.4) running on SPARC station 5 with 64 MBytes of memory, the PAR model resulted in a resource consumption shown in the table below.

time parameters				*resource usage*		
`dK`	`dL`	`dR`	`To`	*transitions*	*memory* [MB]	*CPU time* [sec]
3	3	1	9	764	2.4	0.8
30	30	10	90	2807	2.5	1.0
300	300	100	900	23237	4.4	3.4
3000	3000	1000	9000	227537	21.7	25.9

Spin was verifying the safety properties (like absence of deadlock, and unspecified receptions), and also the assertion that always the expected message was received, by an exhaustive search of the state space.

With the growth of the time parameters the state space increases linearly. Unfortunately, it is only a characteristics of simpler protocols like PAR. For more complex protocols usually an exponential growth is manifested.

When given a PAR program with incorrect time parameters - when $\texttt{To} < \texttt{dK} + \texttt{dR} + \texttt{dL}$ - the validator was able to find the sequence of events that lead to message loss by the aforementioned scenario (Subsection 4.1). This is another useful feature, because, even if the complete validation of the model is impossible, it can help in discovering sequences of events that lead to an incorrect system behavior (if they occur in the early phase of the state space search, of course).

4.3 Some experimental results

We have tested the implementation on various concurrent systems known in the literature (e.g. Train gate controller, Leader election protocol, Seitz circuit, Fischer's mutual exclusion protocol), among which we would like to emphasize the Bounded Retransmission Protocol (BRP) [8, 10], an industry protocol used by Philips. Following the lines from [8] we have been able to obtain all the verification results presented there with memory usage of 4MB. Moreover, our approach has an advantage that Spin treats data more naturally than Uppaal, a tool for symbolic model-checking of real-time systems used in [8]. (Various Promela models are available from the author).

5 Conclusions and Future Work

We presented an implementation of discrete time in Promela and Spin, using integer variables as stop-watches (clocks) to stamp the time slices. The core idea was to use Promela `timeout` predefined statement as a mechanism for process (i.e. clock) synchronization.

The new timed language constructs were implemented as Promela macro definitions in the spirit of the syntax of timing constructions in programming languages. In our opinion, for languages like Promela in which the notion of state is implicit, those transition oriented constructions are more appropriate than formalisms like timed automata, which are more state oriented (especially if we have state invariants). The usage of timing features was demonstrated on the specification and validation of PAR protocol.

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.

It would be very interesting to incorporate the ideas about data and time abstraction of [7] in the existing models in order to obtain verifications that are independent of the concrete parameter values, and to see how general is their applicability.

One possible direction for further research is to establish the relation of discrete time extensions of Promela and Spin with "more formal" formal methods. First natural candidate would be some version of timed Büchi automata with discrete timing.

Another suitable option could be process algebra. In fact, this paper was instigated by the author's work on the Algebra for Communicating Processes with Discrete Relative Timing (ACPdrt) [3], combined with the idea of treating clocks as variables from Clocked Transition Systems [15]. A natural attempt would be to formally connect it with Promela and Spin. One straightforward way is to give a formal semantics of Promela in discrete time process algebra. For that purpose we can use the state operator from [4] which should be adapted to discrete time. It will also be useful to have an automated way to transform the specifications in a reverse direction – from process algebra into Promela – through a possible counterpart of the state operator in the opposite direction.

As we stated in the introduction, this approach can be regarded as a first stage toward an implementation of the discrete timing on the level of Spin validator. First results in this direction are very encouraging The gain in performance is significant, especially with the possibility of using the full power of partial order reduction in the discrete time framework [5] [3]. Nevertheless, because it is essentially the same idea that is implemented both on high- and low-level, the high-level Promela implementation can always be used as a testing platform for the new features before their final implementation in the Spin code.

The concept of using a special daemon process like `Clocks` which acts as a synchronizer and changes the values of non-discrete variables can be further extended for modeling in Spin of dense time based on the timed automata formalism and and even hybrid systems.

For this purpose one should observe that the vectors of integer clock values can be regarded as equivalence classes of a timing bisimulation between the system states. Now, if we replace this equivalence classes for discrete time with equivalence classes for dense time, and allow `Clocks` procedure to "tick" them, we obtain an implementation of region automaton for the corresponding timed automaton [2].

Similarly, the implementation from this paper can be adapted for modeling of a class of hybrid automata, called *discrete time rectangular automata* [11] in a straightforward way - by just allowing the `Clocks` procedure to decrease or increase the value of the analog variables with different rates and for more then one unit per tick, as it is in the present implementation [6].

Acknowledgments: The author would like to thank for the help during the work on this paper: Dennis Dams, Stavros Tripakis, Jos Baeten, Michel Reniers, Sjouke Mauw, Rob Gerth and Bart Knaack.

References

1. Alur, R., *Timed automata*, 1991, Ph.D. thesis, Stanford University
2. Alur, R., Dill, D., *A Theory of Timed Automata*, Theoretical Computer Science, 126, 1994, pp.183-235
3. Baeten, J. C. M., Bergstra, J. A., *Discrete Time Process Algebra*, Formal Aspects of Computing 8 (2), 1991, pp. 142-148
4. Baeten, J. C. M., Weijland, F., *Process Algebra*, Cambridge University Press, 1990
5. Bošnački, D., Dams, D., *Integrating Partial Ordrer Reduction with Fictitious Clocks*, forthcoming paper
6. Bošnački, D., *Modeling Hybrid Systems in Promela and Spin*, forthcoming paper
7. Dams, D., Gerth, R., *Bounded Retransmission Protocol Revisited*, to appear in Second International Workshop on Verification of Infinite Systems, Infinity, 1997
8. D'Argenio, P. R., Katoen, J.-P., Ruys, T., Tretmans, J., *The Bounded Retransmission Protocol Must Be on Time!*, TACAS'97, 1997
9. Gerth, R., Kuiper, R., Peled, D., Penczek, W., *A Partial Order Approach to Branching Time Logic Model Checking*, 1996, to appear in Information and Computation

[3] The executable version of the validator is available from the author

10. 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
11. Henzinger, T. A., Kopke, P. W., *Discrete Time Control for Rectangular Hybrid Automata*, Proceedings of the ICALP'97, LNCS 1256, pp.545-558, Springer-Verlag,1997.
12. 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.
13. Holzmann, G. J., *Design and Validation of Communication Protocols*, Prentice Hall, 1991. Also: `http://netlib.bell-labs.com/netlib/spin/whatispin.html`
14. Kars, P., *Formal Methods in the Design of Storm Surge Barrier Control System*, Hand-outs of School on Embedded Systems, Veldhoven, The Netherlands, 1996
15. Kesten, Y., Manna, Z., Pnueli, A., *Verifying Clocked Transition Systems*, Hand-outs of School on Embedded Systems, Veldhoven, The Netherlands, 1996
16. Kluesner, 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. Tanenbaum, A., *Computer Networks*, Prentice Hall, 1989
19. Tripakis, S., Courcoubetis, C., *Extending Promela and Spin for Real Time*, TACAS '96, LCNS 1055, Springer Verlaag, 1996
20. Vaandrager, F., *Two Simple Communication Protocols*, in Baeten, J.C.M., ed., Applications of Process Algebra, pp.23-44,Cambridge University Press, 1990