Copyright © IFAC Real Time Programming,
Gramado, RS, Brazil, 1997

MODELLING AND ADA IMPLEMENTATION OF REAL-TIME SYSTEMS USING TIME PETRI NETS

F. J. García Izquierdo* **and J.L. Villarroel Salcedo†**

* *Universidad de La Rioja, Dpto. Matemáticas y Computación, C/ Luis de Ulloa s.n., 26004 Logroño, Spain*
† *CPS, Universidad de Zaragoza, Dpto. de Informática e Ing. de Sistemas, C/ Maria de Luna 3, 50015 Zaragoza, Spain*

Abstract. Time Petri Nets are used as the formalism for the developing of the whole life cycle of real-time systems. We comment on how to model real-time systems using this formalism and we focus our work on the automatic code generation for these systems. We present a technique for the automatic generation of the code skeleton (control part). Finaly we assess the performance of the implementation.

Keywords. Time Petri Nets, Automatic Code Generation, Ada Tasking, Real-Time.

1. INTRODUCTION

The difficulty for analysing and building up real-time systems is well-known. Moreover, in many real-time systems both software and hardware reliability are critical aspects due to the possible catastrophic effects that a failure can produce. Focusing on software as our main aim, the use of an automatic tool in the coding phase of the life cycle will prevent us from making coding mistakes, it will simplify the development of the system and, therefore, reduce significantly its cost. Moreover, the use of formal methods in the whole development cycle can fulfil the previously mentioned reliability requirement, since these kinds of methods can allow the verification of functional and temporal requirements.

Petri Nets have been widely used for modelling and analysing discrete event systems because of features like the possibility of modelling concurrence, resource sharing, synchronizations, ... Moreover Petri Nets have a strong mathematical foundation which allows the validation and verification of a wide set of correctness and liveness properties. In this paper we assume that the reader knows the basic concepts of Petri Nets (see Murata, 1989 for a survey). However, classical Petri Nets are not suitable either for the modelling or the analysis of real-time systems, due to the impossibility of including time features and constraints in the model. To avoid these problems many authors have proposed extensions which add time characteristics to the basic Petri Nets. The first of these extensions, *Timed Petri Nets* (Ramchandani, 1973), considers a deterministic and fixed time value associated to each transition in the net, modelling the duration of the firing of the transition, which can be seen, as an initial approach, as the duration of some activity. This kind of nets does not consider the fact that the duration of some tasks in real-time systems is not fixed, and depending on the system or environment state takes different values. One way for the modelling of this feature is to associate a random duration (sometimes called delay) with some probabilistic distribution, for the firing time of the transitions. This leads to the *Stochastic Petri Nets*. These nets have been commonly used for performance evaluation, analysis of parameters such as throughput, service time, number of tasks in the system, ... All of them are mean values of the parameters.

Nevertheless an stochastic treatment for a real-time system is highly unadvisible since it is impossible to guaranty absolute time properties. To avoid these problems *Time Petri Nets* (Merlin and Faber, 1976; Berthomieu and Diaz, 1991) were proposed. In Time Petri Nets an interval specifying an upper and a lower bound for the duration of the task is associated to each transition. This approach is more general than Timed Petri Nets, since these can be modelled using a Time Petri Net, but the opposite is not true. The bounds associated to a transition can be used to verify time properties and constraints. Time Petri Nets were initially defined by Merlin and Faber, 1976, and other authors have proposed different forms to integrate temporal intervals in the Petri Nets formalism: *Place/Transition Nets with Timed Arcs* (Hanisch, 1993); *High-Level Timed Petri Nets* (Felder, Ghezzi and Piezzé, 1993); *Interval Timed Coloured Petri Nets* (van der Aalst, 1993). The latter

considers tokens of different colours (representing different kinds of resources, tasks, ...) and with an associated timestamp (time when the token was generated) which is considered while studying the enabling of the transitions. Considering only temporal aspects, we have chosen Time Petri Nets (hereafter TPN) due to its expressive power being equal to or greater than the rest of the extensions of Petri Nets and, from our point of view, it is more intuitive and suitable for the specification of the systems which are the object of our study. In no case do we consider using High Level Petri nets in these fisrt stages of the research project.

TPNs are useful in order to develop reliable real-time software due to the possibility of modelling timeouts, periodical activities, synchronizations, concurrency, ... Using the same formalism along the life cycle will allow the detection of bad properties and malfunctions in the early stages of the cycle. The use of a formalism like TPNs allows us not to restrict the structure of systems in order to analize their temporal constraints. In this sense, the design flexibility is increased with respect to the use of classical analytic techniques such as *Rate or Deadline Monotonic Analysis*. In these approaches, for example, in order to allow the analysis, the communications between the periodical tasks must take place through an intermediate server with no guarded entry. The use of TPNs for the analysis of real-time systems eliminate these kinds of restrictions. Moreover, the use of this formal method will allow us the automatic code generation (as it will be seen in this paper), and, therefore, the avoidance of making mistakes during the codification stage.

This paper is the first of a set whose main aim is the automatic code generation for Real-Time systems. The scope of this paper is limited to the modelling and the implementation of real-time systems. A software implementation of a system is a program that satisfies every functional requirement of the system. With reference to Petri Nets, an implementation is a program which simulates the firing of the net transitions, observing the marking evolution rules. The system is modelled using a TPN and latter implemented. An adaptation of classical Petri Net implementation techniques, which adds time information to the net transitions, is used. As for classical implementation techniques (Colom, Silva and Villarroel, 1986) we can distinguish between compiled and interpreted, with the compiled implementations being divided into centralized and distributed. The former use a single coordinator process responsible for the control and evolution of the net. There exist several techniques like *guards*, *representing places* or *enabled transitions*. The distributed implementations split the control between several processes, each one implementing a subnet. In this paper we only consider monoprocessor systems and our implementation will be a modification of the centralized representing places and enabled transitions techniques. Ada 95 is used as the language for the implementation code.

This paper is organized as follows: section 2 shows how to model real-time systems using TPNs. In section 3 we present the implementation technique. Section 4 shows some conclusions and outlines future work.

2. MODELLING REAL-TIME SYSTEMS USING TIME PETRI NETS

We use TPNs to model systems consisting of a set of concurrent activities with temporal constraints, i.e. real-time systems. We can model periodic or aperiodic processes, which communicate with each other, with synchronizations (rendezvous, ...), timeouts, exceptions, ... We can see TPNs as Petri Nets with labels: two time values (α_i, β_i) associated to transitions. The first time value represents the static *Earliest Firing Time* (static EFT), the minimum time, starting from t (time at which t_i is enabled), that a transition has to wait until it can be fired, and the second is the static *Latest Firing Time* (static LFT), the maximum time that a transition can be enabled without firing. Assuming that transition t_i was enabled at time t, and is being continuously enabled, these two time values allow the calculation of a firing interval for each transition t_i in the Net. The firing of t_i must occur in the interval $(t+\alpha_i, t+\beta_i)$. Once the transition is to be fired, the firing is instantaneous. For a formal description about Time Petri Nets see for example Berthomieu and Diaz, 1991. Unfortunately there are still few results applicable to TPNs. Analysis of TPNs mainly uses enumerative methods which involve the computation of the *reachability graph* (see Berthomieu and Diaz, 1991; Popova, 1991).

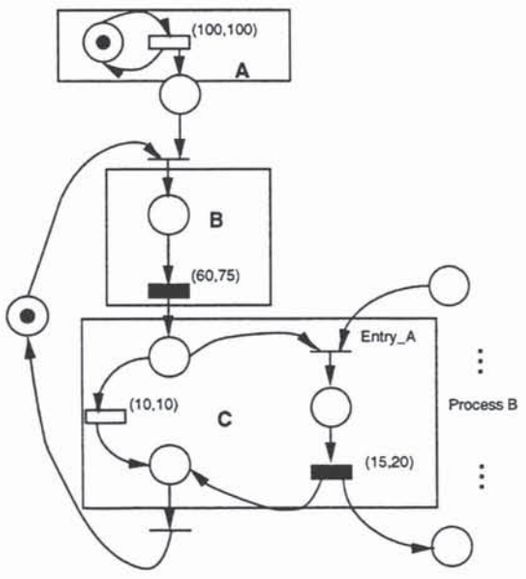

Figure 1: Example of model

```
loop
  CODE;                        -- Box B
  select
    Process_B.entry_A;         -- Box C
```

```
      or
        delay 10.0;
      end select;
      delay until Next;          -- Box A
      Next := Next + 100.0;
    end loop;
```

For example, in fig.1. we can see a TPN model of a periodic process that executes a piece of code and communicates with another process. This communication has an associated timeout. Three elements in fig.1 have been highlighted (a piece of Ada code with the same behaviour of the TPN model is provided for a better understanding of the model):

- Box B shows an action, i.e. *code*, to be executed by the process. The execution starts when the input place becomes marked. The execution must finish at a time between (60, 75), i.e. the computation time of the code is between (60,75) time units. When the execution ends, the transition is fired.
- Box A models the *periodic activation of the process.* Every 100 time units the transition fires and promotes the execution of the process.
- Box C shows a *timeout in a communication with another process.* Let us suppose that the place is marked at time t. If the transition labelled with entry_A does not fire (starts the communication) before $t+10$ (expiration time of the timeout), then transition (10, 10) will fire, aborting the starting of the communication.

Transitions in Merlin's model are all of the same type. They all have the same functionality. But in a real-time system there are different situations that are suitable to be modelled as a transition. In order to highlight in our models the different roles that a transition can play and with the aim of implementing the model we distinguish three kinds of transitions:

- *CODE-Transitions* (CODE-T). One of these transitions, together with its input place, represents the code associated to one activity. This activity starts its execution when the enabling of the transition is detected. These transitions are tagged with two time values (α, β), in accordance with TPN fashion. In the model, the meaning of these time values is associated to the execution time of the activity. At best, the code execution will have finished at time α, and at worst the execution will last β. Thus, the execution takes a time between (α, β). The finalization is represented by the transition firing. We draw a CODE-T as a thick segment.
- *TIME-Transitions* (TIME-T) are transitions with an associated time event, e.g. a timeout or the next periodical activation of a process. These transitions also have associated time information, described with an interval (α, α), where α represents the event time. The firing of this kind of transitions represents the occurrence of the event, which causes control actions to take place on the system. If a timeout related to an action occurs, the action must be aborted and the resources used by it released. If a periodic activation event occurs, the related periodic process must start its execution again. We draw a TIME-T as an empty thick segment.
- *SYCO-Transitions* (SYCO-T) are transitions with no temporal meaning. They are used to perform synchronizations (SY) and control (CO) tasks. The firing of a transition of this kind leads to plain state changes or synchronizations among activities. We draw a SYCO-T as a thin segment.

We also need to impose several structural restrictions to the net, so that it can model an actual situation:

- Each place may have simultaneously, at most, a CODE-T, a TIME-T and several SYCO-Ts as output transitions. In this way, we avoid conflicts between CODE-Ts or TIME-Ts (since this situation has no meaning in this field). Conflicts may appear between SYCO-Ts.
- Each CODE-T has a single input place, making up a functional unit of code. This place must be 1- bounded, because we only consider sequential processes without replication capabilities.
- The time values associated to TIME-Ts must be the same, i.e. (α, α). These time values represent the time occurrence of an event. An interval like (α, β) has no meaning in this context.
- SYCO-Ts have no time meaning and, therefore, they have no associated time interval, according to the TPNs formalism.

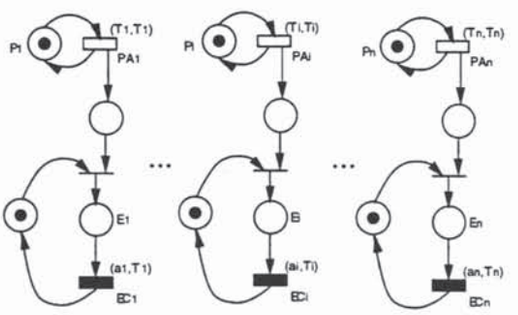

Figure 2: Example of N periodical process

As a simple example (which we will consider in the test of the performance of the implementation) we can present in Fig. 2 the model of a real-time system made up of several periodical processes, with no communication between them. The net shows a set of marked places $\{P_i\}$ joined to a corresponding set of TIME-Ts $\{PA_i\}$, which model a set of periodical activations of processes. Note that the time tags are (T_i, T_i), i.e., the period of each process. The task performed by the processes is modelled with places E_i and EC_i, representing the execution code of the process. This code has a minimum duration a_i and will last at most until the next periodical activation T_i, i.e., the deadline is equal to the period. Every T_i

time units a new mark representing a new periodical activation of the process is generated by PA_i.

3. THE IMPLEMENTATION

An implementation is a program which simulates the firing of the net transitions, observing the marking evolution rules. We attempt the implementation of the system modelled using a TPN. We use centralized techniques. It is assumed that a *scheduling* for the processes in the real-time system has been computed off-line. This scheduling guarantees that all processes in the system finish their execution before their deadlines. The effect of the scheduling is to assign a static priority to each transition in the net. These priorities are taken into account in two different ways. The first is to assign static priorities to the Ada-tasks which represent the firing of the transitions which have an associated code. These priorities are used by the Ada-Kernel to schedule the different tasks. The second is used by the net coordinator to solve the conflicts among transitions. The highest priority transition will always be chosen to fire. We use Ada 95 as the implementation language and the preemptive mechanism provided by the Ada Kernel.

As in any centralized implementation, we consider two types of processes (hereinafter we will talk about Ada-tasks when we refer to a process):

- Each unit {place + CODE-Transition}, which represents each code execution of the net, will be implemented as an Ada-task, called *CODE task*. In this way we maintain the concurrency of the model in the implementation. Each CODE task will execute the code associated to the unit. This code must be developed separately, and later linked together.
- Control and timing supervision will be performed by another task called *coordinator*. Every CODE task communicates with the coordinator, which is responsible for taking decisions as to when a transition must fire. It will be the highest priority task in the system, since control and timing actions must be performed immediately. We can see the coordinator as the kernel of an operating system, and the CODE tasks as the processes managed and executed in it.

The remaining transitions of the net (TIME-Ts and SYCO-Ts) are considered by the coordinator to perform the control of the net implementation. In this way, the operational part of the system is performed by the CODE tasks and the control part, by the coordinator.

Several choices related to the firing of transitions in conflict must be made for the implementation. A place may have at most, one CODE-T, one TIME-T, and several SYCO-Ts output transitions. If there is a conflict situation, i.e. several output transitions of the same place are enabled, we choose one SYCO-Ts to be fired, since they represent control actions in the net and they must take place immediately. In this choice the transition priority is considered. If there is no enabled SYCO-T we will execute the associated CODE task and record the time of the event modelled with the TIME-T, if it exists. A race policy is adopted to decide which is the actual transition selected to fire; e.g.: if the event (TIME-T) expires before the execution of the CODE task ends, the TIME-T must be fired and the code execution must be aborted. This models the occurrence of a timeout. The opposite case is also possible.

To describe the working of the implementation we only show the behaviour of the coordinator. The control structure of the coordinator is based on the execution of an infinite loop. In this loop, the coordinator performs several actions related to the net control and time monitoring. At each execution of the loop the coordinator examines every transition of the net and determines if it is enabled, proceeding to its firing or to the performance of the requiered actions. SYCO-Ts are tested firstly. TIME-Ts are tested secondly and the associated events recorded in an event list. CODE-Ts are finally tested and, if enabled, the start rendezvous with the associated CODE task is accepted, allowing the CODE task to begin execution. The actual firing of TIME-Ts and CODE-Ts is postponed. Once the coordinator has fired every firable SYCO-T, has registered every time event and has accepted every CODE task start, the coordinator must wait for the end of a CODE task or the expiration of an event. When a CODE task ends or an event expires, then the corresponding CODE-T or TIME-T must be fired. If a CODE task ends the coordinator must accept an end rendezvous with it. When a transition is fired, the marking of the net is updated. This leads to new enabled transitions and must provoke reexecution of the coordinator loop, and so on.

We have implemented the nets using two centralized techniques: the *representing places technique* and the *enabled transitions technique*. They differ from each other in the way in which they evaluate the enabling of a transition. We give a brief explanation of both techniques (for further details refer to (Colom, Silva and Villarroel, 1986)). The former chooses one of the input places of every transition as its representing place, in such a way that only transitions whose representing place is marked are taken into account for the enabling test. The latter uses a counter for every transition whose value is the number of unmarked input places of the transition. In this way, a transition is marked if the value of its counter is zero. This treatment simplifies the enabling test causing the performance of the enabled transitions technique to be better than the other. Therefore we will deal with it during the rest of the paper.

Several data structures are necessary for the enabled transitions implementation: `Net_STs`, `Net_TTs` and `Net_CTs`, which record for each kind of transition (STs, Syco; TTs, Time; CTs, Code) the characteristics of every transition of the net: number of input places, a list of transitions of each type in

conflict with it, a list of descending transitions of each type (that may be enabled by means of the firing of the present one), temporal bounds (EFT,LFT) for CODE-Ts, firing time (FT) for TIME-Ts, and priority for SYCO and CODE transitions. Three arrays containing the counters for every transition in the net are also needed. They represent the marking of the net. One stack containing the enabled transitions is provided for each kind of transitions: SYCO_PL, TIME_PL and CODE_PL, where PL means processing list.

An implementation of a TPN can provoke the appearance of an accumulative temporal shifting due to the difference in time between the time when a transition gets enabled and the time when the coordinator realizes the transition is enabled. This problem has been solved using a time variable (Last_Update) to record the time when the last marking update was produced. Doing this we can avoid having to associate a timestamp to the tokens, as in other proposed models (van der Aalst, 1993; Felder, Ghezzi, Piezzé, 1993).

With these comments we can present the code of the coordinator for the enabled transitions technique:

```
task body Coordinator is
  -- declaration of variables and subprograms
begin
 accept Coordinator_Start;
 Last_Update := CLOCK;
 loop
   -- Study enabled SYCO-T and start firing
  while not Empty(SYCO_PL) loop
    if Enabled_STS(Top(SYCO_PL)) = 0 then
     Demark_In_Ps_ST(Top(SYCO_PL));
    end if;
    Pop(SYCO_PL);
  end loop;
   -- Study enabled TIME-T. Recod events
  while not Empty(TIME_PL) loop
   if Enabled_TTS(Top(TIME_PL)) = 0 then
    Record_Event(Top(TIME_PL), Last_Update
     + Net_TTS(Top(TIME_PL)).FT);
   end if;
   Pop(TIME_PL);
  end loop;
   -- Study enabled CODE-T. Start execution
  while not Empty(CODE_PL) loop
   if Enabled_CTS(Top(CODE_PL)) = 0 then
    accept Start(Top(CODE_PL));
   end if;
   Pop(CODE_PL);
  end loop;

  loop
   select
    accept End_Task( T : in CT ) do
       -- CODE-T firing
     Last_Update := CLOCK;
     Demark_In_Ps_CT( T );
     Mark_Out_Ps_CT( T );
    end End_Task;
   or
    when not Event_List.Empty =>
     delay until Urgent_Event_T;
      Last_Update := Urgent_Event_T;
      loop
       if Urgent_Event_Is_Time then
         -- TIME-T firing
        Demark_In_Ps_TT(TT_Urgent_Event);
        Mark_Out_Ps_TT(TT_Urgent_Event);
       else
          -- finish SYCO-T firing
        Mark_Out_Ps_ST(ST_Urgent_Event);
       end if;
       Pop_Event;
       exit when Event_List.Empty or else
         Urgent_Event_T /= Last_Update;
      end loop;
   end select;
   exit when End_Task'COUNT /= 0;
  end loop;
 end loop;
end Coordinator;
```

Tasks Transitions execute their associate code after a *start rendezvous* with the coordinator and, when finishing, another *end rendezvous*.

```
task body CODE_Task is
 Ident: CT;
begin
    -- Initialization
 loop
  Coordinator.Start (Ident);
  Code (Ident);
  Coordinator.End (Ident);
 end loop;
end CODE_Task;
```

Demark_In_Ps_?? and Mark_out_Ps_?? are procedures which update the marking of the net, i.e. update the counters of the descending transitions of the fired transition, and abort the possible transitions in conflict with the fired transition.

3.1 Performance evaluation

Every control and timing action that the system must accomplish is performed by the coordinator. Each time a transition is fired, i.e. a code is executed or an event is expired, the coordinator acts and promotes the firing of the remainder of the net. So the coordinator introduces an overload into the system which reduces the maximum schedulable utilization (U). In order to evaluate the impact of the coordinator we have run an experiment consisting in the implementation of net modelling a system made up of N periodical tasks, all of them of the same period. Theoretically, the schedulable utilization must be 100%, but if we consider factors such as the operating system, the delay accuracy, the context switching and other factors due to the Ada kernel, this limit is actually reduced to 94% in the worst case. We vary the number of processes (from 5 to 30) and their period (from 1 to 0.2 seconds) and register the maximum schedulable utilization for each case. This leads us to the following table. The figures have been calculated for the SPARC CPU -5V with the microSPARC-II processor, GNAT 3.03 Ada95 for Solaris compiler; similar results have been obtained for other platforms. The figures include the overload due to the coordinator and to the Ada kernel.

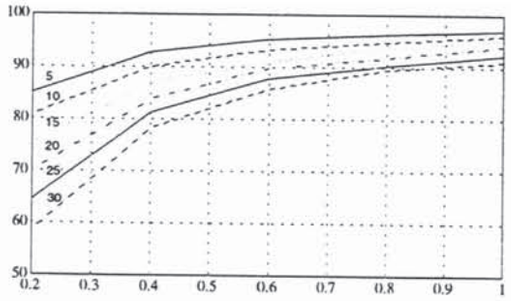

Figure 3: Schedulable utilization U(%)

Table1. U(%) function of period and n. of tasks

	1	0.8	0.6	0.4	0.2
5	96.92	96.17	95.17	92.60	85.02
10	95.83	94.75	93.11	89.81	80.59
15	95.03	93.55	90.93	86.80	75.28
20	93.90	91.69	89.60	83.61	70.00
25	92.05	90.16	87.67	81.18	64.53
30	91.02	89.51	85.65	78.28	58.41

4. CONCLUSIONS AND FUTURE WORK

We have proposed the use of a formalism for the whole life cycle of real-time systems, which allows the modelling of all the situations appearing in these systems: Time Petri Nets. This bring us the following advantages: in the first place it is a formalism that allows an unambiguos system specification and easy to understand due to its graphical nature; its use will help us during the development of the system since it allows the verification and validation of the correction of the system in the early stages of the cycle; the analysis capabilities of Time Petri Nets allows a high modelling flexibility, since it will no longer be necesary to impose restrictions on the system in order to analyze the temporal behaviour and verify the timing constraints; moreover, this formalism is executable, and therefore it allows the prototyping and simulation of systems.

We have also proposed an automatic code generation technique, for systems modelled with the chosen formalism. We can automatize the coding phase, preventing us from making coding mistakes, simplifying this stage, and so, reducing the cost of the system. It is an interpreted technique, so it allows us to modifify with ease the control structure of the system without changing the application tasks.

However, we have detected some problems. The presence of the coordinator introduces an overload into the system which reduces the maximum schedulable utilization, as we have commented before. On the other hand, we are dealing with a centralized technique. This means that the produced implementation is sensitive to faults, since if the coordinator fails the whole system fails.

In future works we will try to evaluate the overload introduced by the coordinator for different net topologies and classes, and deduce what is the topology and class for which the implemetation works better. We will also develop distributed techniques for the implementation, which avoid the use of the coordinator, eliminating the above mentioned overload and making the system more fault tolerable, since the control tasks are split between several processes, one of which can fail without implying the total inavailability of the system.

ACKNOWLEDGEMENTS

This work has been supported in part by project TAP94-0390 from the Comisión Interministerial de Ciencia y Tecnología of Spain and project PIT-11/93 from the Gobierno de Aragón.

REFERENCES

Berthomieu, B. and M. Diaz (1991). Modeling and Verification of time dependent systems using time petri nets. IEEE transacctions on Software Engineering, 17(3):259-273, March 1991.

Colom, J.M., M. Silva, J.L. Villarroel (1986). On software implementation of petri nets and colored petri nets using high-level concurrent languajes. In Proc of 7th European Workshop on Application and Theory of Petri Nets, pages 207-241, Oxford, July 1986.

Felder, M., C. Ghezzi, and N. Piezzé (1993). High-Level Timed Petri Nets as a Kernel for Executable Specifications. Real-Time Systems, 5, 235-248.

Hanisch, H.-M. (1993). Analysis of Place/Transition Nets with Timed Arcs and its Application to Batch Process Control. In: Proc. of Applications and Theory of Petri Nets, Chicago, Illinois, USA, June 1993 (Marco Ajmone Marsan (Ed.)). pp. 282- 299Springer-Verlag

Merlin, P. and D.J. Faber (1976). Recoverability of communication protocols. IEEE transactions on Communication, 24(9), September 1976.

Murata, T. (1989). Petri Nets: Properties, Analysis and Applications. In Proc. of the IEEE, vol. 77, N. 4, pp. 541-580, Apr.1989.

Popova, L. (1991). On Time Petri Nets. J. Inform. Process. Cybern., vol EIK 27, N. 4, pp. 227-244.

Ramchandani, C. (1973). Performance Evaluation of Asynchronuos Concurrent Systems by Timed Petri Ntes, PhD thesis, Massachusetts Institute of Technologiy, Cambridge 1973.

van der Aalst, W.M.P. (1993). Interval Timed Coloured Petri Nets and their Analysis.In Applications and Theory of petri Nets, Chicago, 1993, M. Ajmone Marsan, eds, pp. 451-472, Springer-Verlag, Berlin.