DECENTRALIZED IMPLEMENTATION OF REAL-TIME SYSTEMS USING TIME PETRI NETS. APPLICATION TO MOBILE ROBOT CONTROL.

F.J. García* J.L. Villarroel**

* *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 whole life cycle of real-time systems. We present how to model real-time systems using this formalism and we focus our work on the code generation for these systems. The first step of this implementation technique consists of the extraction of the processes (states machines) embedded in the net, each of which is implemented in an Ada task. The result of the application of this technique is a set of concurrent processes coupled by means of synchronous or asynchronous communications, with the same behaviour as the model.

Keywords: Formal methods, Petri-nets, modeling, real time, Ada tasking

1. INTRODUCTION

In this paper we continue the work started with García and Villarroel (1996) in which a way of modeling real-time systems (using time Petri nets) and an automatic technique (through the use of an interpreter) for code generation was shown. Our objective was and is to improve the reliability in all steps of the life cycle of a real-time system, by means of a unique formal method during it: time Petri nets.

Petri nets have been widely used for modeling and analyzing discrete event systems because of the possibility of modeling concurrency, resource sharing, synchronizations, ... However, ordinary Petri nets are not suitable either for the modeling or the analysis of real-time systems, due to the impossibility of including time features in the model. This is the reason why in this paper a time extension of Petri nets, Time Petri Nets (TPN), is used. TPNs are useful in order to develop reliable real-time software due to the possibility of modeling time-outs, synchronizations, concurrence, periodic or aperiodic communicating processes.

Other references can be found in the literature where time Petri nets are used in relation to real-time systems. Some times they are used for the modeling and analyzing communication or control systems (Berthomieu and Diaz (1991); Buy and Sloan (1994); Aalst and Odijk (1995)). On other occasions (Shatz et al. (1996)) Petri nets are used to generate nets models of concurrent programs (tasking programs) with the purpose of an analysis of the concurrent behaviour, especially deadlock properties. Finally another line of research (Gedela and Shatz (1997)) has modeled Ada-tasking structures with time Petri nets as a way of defining precise behaviour for tasking semantics and providing support for automated analysis.

In this work we consider just the opposite approach: TPN are considered as the initial tool for the modeling of the real-time system. After analysis and validation, the implementation is generated from the TPN model, being this last point the main contribution of this paper. The use of the same formalism during the life cycle will allow the detection of bad properties in the early stages of the cycle and, indeed, will allow us not to restrict the structure of systems in

order to analyze their temporal constraints. In this sense, the design flexibility is increased with respect to the use of classical analytic techniques such as RMA where, 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. Moreover, the use of this formal method can allow us the automatic code generation (García and Villarroel (1996)), and so it avoids making mistakes during the codification.

The scope of this paper is limited only to the modeling and the implementation of real-time systems. A Petri net implementation is a program which simulates the firing of the net transitions. Adaptations of Petri net classical implementation techniques are used (Colom et al. (1986)) which can be split into centralized and decentralized ones. The former use a single coordinator process responsible for the control and firnig of the transitions of the net, which represent the operational part of the system. This technique (object of study in the related paper García and Villarroel (1996)) is subject to several problems related to the presence of the coordinator, which acts in every transition firing introducing an overload into the implemented system. In addition, the coordinator alone is responsible for the control of the implementation, sequentializing the control of the implemented system, which is in fact concurrent, and making it sensitive to faults, since if the coordinator fails the whole system fails too. The decentralized implementations try to solve these problems by splitting the net into several concurrent subnets. Ada 95 is used as the language for the implementation code and only monoprocessor platforms are considered.

2. MODELING REAL-TIME SYSTEMS USING TIME PETRI NETS

A Time Petri Net (Berthomieu and Diaz (1991)) is a tuple $(P, T; F, B, M_o, SIM)$, where $(P, T; F, B, M_o)$ defines a marked Petri net, the *underlying Petri net*; and SIM is the mapping called static interval $SIM : T \to \mathbb{Q}^* \times (\mathbb{Q}^* \cup \infty)$, where $\mathbb{Q}^*$ is the set of positive rational numbers. Thus, TPNs can be seen as Petri nets with labels: two time values (α_i, β_i) associated to transitions. Assuming that transition t_i was enabled at time θ_0, and is being continuously enabled, the first time value represents the minimum time, starting from θ_0 that t_i has to wait until it can be fired, and the second is the maximum time that t_i can be enabled without firing. So these two time values allow the calculation of the firing interval for each transition t_i in the net: $(\theta_0 + \alpha_i, \theta_0 + \beta_i)$. Once the transition is to be fired, the firing is instantaneous.

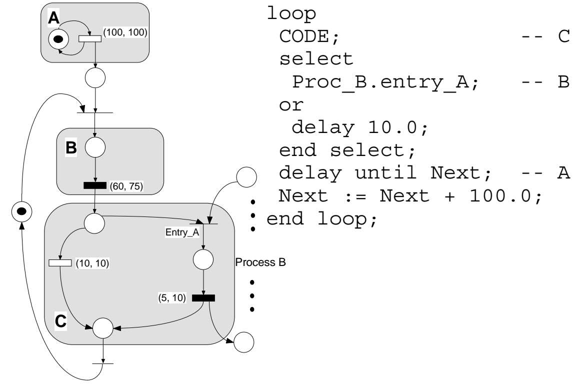

Fig. 1. Example of TPN model

As an example in fig.1, a TPN modeling a periodic process that executes a piece of code and communicates with another process is shown. This communication has an associated time-out. Three elements have been highlighted (a piece of Ada code with the same behaviour 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 gets marked. The computation time of this activity is between (60,75) time units.
- 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 communication with another process which has an associated time-out. Let us suppose that the place is marked at time τ. If the transition labeled with `entry_A` does not fire (starts the communication) before $\tau + 10$ (expiration time of the time-out), then transition (10,10) will fire, aborting the starting of the communication.

Transitions in TPN have the same functionality. But the different situations that appear in a real-time system must be highlighted in our models. Therefore, and with the aim of implementing the model, we distinguish three kinds of transitions:

- CODE-Transitions (thick segments) together with its input place, represents the code associated to an activity, that starts its execution when the transition gets enabled, i.e. the input place gets marked. The two time values (α, β) represent the execution time of the activity. At best, the code execution finishes at time α, and at worst the execution will last β. The firing represents the end of the code.
- TIME-Ts. (empty thick segment) are transitions with an associated time event, e.g. a time-out. 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.

- SYCO-Ts. (thin segment) are transitions with no temporal meaning used to perform synchronizations (SY) and control (CO) tasks. The firing of a SYCO-T leads to plain state changes.

3. NET DECOMPOSITION

The idea of decentralized implementations is quite simple. In order to avoid the problems of centralized implementations (mentioned in the introduction) the use of the centralized coordinator must be avoided. Therefore the control of the net is split into several sequential subnets, each of which is implemented in a separate process, concurrent with the others. The identification of the concurrent processes embedded in the net and their inter-connection through a communication mechanism like a buffer or a rendezvous is the first step in the implementation. The basis is to merge into a single process a set of transitions in mutual exclusion (ME) with the others in the subnet (two transitions t_i, t_j are in mutual exclusion, $t_i MEt_j$, if they can not be fired simultaneously). A set of transitions which are in ME relationship are not concurrent, so they can be in the same process (π_i) without reducing the actual concurrence. For the computation of the sets of transitions in mutual exclusion the net without time information will be used. The reason is that two or more transitions which are in ME in the Petri net, remain in ME in the TPN, but the opposite it is not true. The application of temporal mutual exclusion is still under research. The decomposition techniques used in this work are based on Colom et al. (1986); Villarroel (1990). Through the computation of compatibility classes of transitions in ME a partition of the net is obtained solving a coverability problem. Each class is a set of transitions in ME that can be implemented as a sequential process (π). The ME of transitions can be achieved by a computation of *monomarked p-invariants*. Monomarked p-invariants are particularly interesting because they describe a set of places in ME. In fig. 2 either p_1, p_2, p_3 or p_4 is marked, but never two or more of them at the same time. Obviously, a set of places in ME implies a set of transitions in ME, the input and output transitions of the places. Unfortunately it is not always possible to cover a Petri net with a set of monomarked p-invariants. To solve this problem Villarroel (1990) proposes a technique (based in the concept of *pipeline*) to make a set of monomarked p-invariants which cover all transitions of the net.

A place p, with respect to a process π_i, can be either *private* (every input and output arc of p

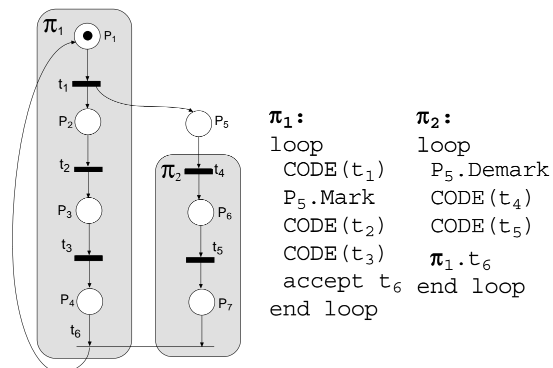

Fig. 2. Decomposition into two sequential processes. An asynchronous comm. in p_5, and a synchronous communication in t_6.

are connected to transitions belonging to π_i), or external (p is only connected to transitions not belonging to π_i), or shared (p is connected both to transitions belonging and not belonging to π_i). In fact, a shared place is modeling an asynchronous communication between two processes. A shared transition represents a synchronous communication (rendezvous). Moreover, it is possible to share sets of transitions and places grouped in a subnet, which are representing the execution of a piece of code in a rendezvous.

4. PROCESS IMPLEMENTATION

However it is achieved, a partition of the Petri net which covers every transition of the net can be found. The partition is made up of a set of sequential processes, each one having an associated p-invariant that can be used to describe the control flow of the process. In this way, only those transitions whose input place belonging to the p-invariant is marked, are able to fire. Each process can be implemented in an Ada task, using a `case` structure. Each place of the p-invariant describes a state which will be implemented at each branch of the `case`. The code associated with each branch depends basically on the output transition of the place (or transitions if there is a conflict). It is possible to improve the implementation using the single token of the p-invariant as if it were the program counter of the process. The flow of the token through the p-invariant defines the execution order of the transitions, avoiding the use of the case structure (e.g. see the code shown for the net in fig.2). Implementing each process in an Ada task, both control and operational parts of the set of transitions are integrated in the same process avoiding the use of a centralized coordinator.

The three kinds of transitions in our modeling approach will be implemented in a different manner. A SYCO-transition will be taken into

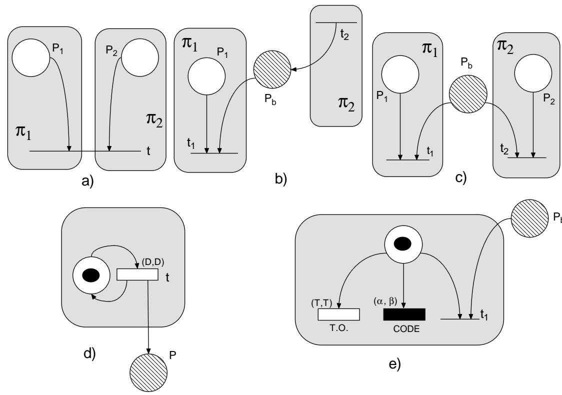

Fig. 3. a) Synchronous communication, transition t; b) asynchronous communication, place p_b; c) simple conflict; d) Periodical activator; e) Inner conflict inside a process;

account to make decisions inside a process or to perform synchronous communication between processes. A CODE-T involves the execution of their associated code. A TIME-T represents a delay in the execution of the process. As a first approximation the delay starts when the input place of the transition gets marked. When the delay expires the transition is fired. But this situation can provoke accumulative drift in the processes, e.g. due to preemption. To avoid it a time variable is associated with each process with TIME-Ts inside. This variable (`Last_update`) records the time at which the last marking update occurred in the process. This time is used in the computation of the expiration time of the delays. E.g., consider these implementations for the fig.3.d. (a periodical activation). The implementation on the left presents accumulative drift, solved on the right.

```
loop          Last_update := CLOCK;
 delay D;     loop
 P.Mark;       next := Last_update + D;
end loop;      delay until next;
               Last_update := next;
               P.Mark;
              end loop;
```

Sometimes a transition or subnet can be shared between two processes. This situation represents a synchronous communication between both processes, and will be implemented with an Ada rendezvous (fig.3.a). The remaining non private places that are not included in a p-invariant act as asynchronous communications between processes that will be implemented with a buffer or a relay process (fig.3.b). There are likely to be several transitions at the output of a place (conflict situation). If the transitions belong to different processes, the place will act as a shared data between them. The descending processes will compete for the token of the place (fig.3.c). This situation is very common, since it is the natural way of modeling shared data or resources. Since there are two or more processes involved in the synchronous or asynchronous communication, in a general case each one can be marked at a different time, thus, it is necessary to communicate the time `Last_update` of the processes.

If the place originating the conflict belongs to a p-invariant all its output transitions in conflict will be implemented in the same process. The simplest case is when several SYCO-Ts depend on the same place. This situation represents a choice, implementable with an `if` structure if there is no communication involved or with a `select` structure with several `accept` branches if there are communications. Other conflicts can appear between different kinds of transitions, as in fig.3.e, which models the execution of a code abortable by the expiration of a time-out, or a control action external to the process. For the implementation of this kind of conflicts we will use the Ada A.T.C. structure.

5. AN EXAMPLE

The technique presented in this paper, has been used for the code generation for the controller of a real-time mobile robot navigation application. Off-line planned trajectories and motions are modified in real-time to avoid obstacles, using a reactive behaviour. The information about the environment is provided to the control system of the robot by a rotating 3D laser sensor with two degrees of freedom.

The controller must perform three main activities: motion control, supervision and data sensor processing. The control loop has a sample period T=0.25 s obtained in the analysis phase so that the system meets all the temporal constraints. The communications between the robot and the controller have defined time-outs: 0.1 s in position reading and 0.1 s in setpoint sending. The objectives of the supervisor activity are: trace the real trajectory of the robot, test if the actual goal has been reached, update the actual goal point and manage the system alarms. At this point of application development, only alarms related to the communication time-outs have been taken into account. The firing of a communication time-out must stop the system within 0.1 s. The 3D laser sends a new scan each 0.1s. The controller must be capable of accepting and processing the sensor data at this rate. The communication with the laser has a 0.2 s time-out. Fig.4 shows the TPN that specifies the control and time restrictions of the real-time system (time information has been removed for clarity). This Petri net has been used in the analysis for the validation of system specifications and for scheduling the controller tasks in the processor by priority assignment (not

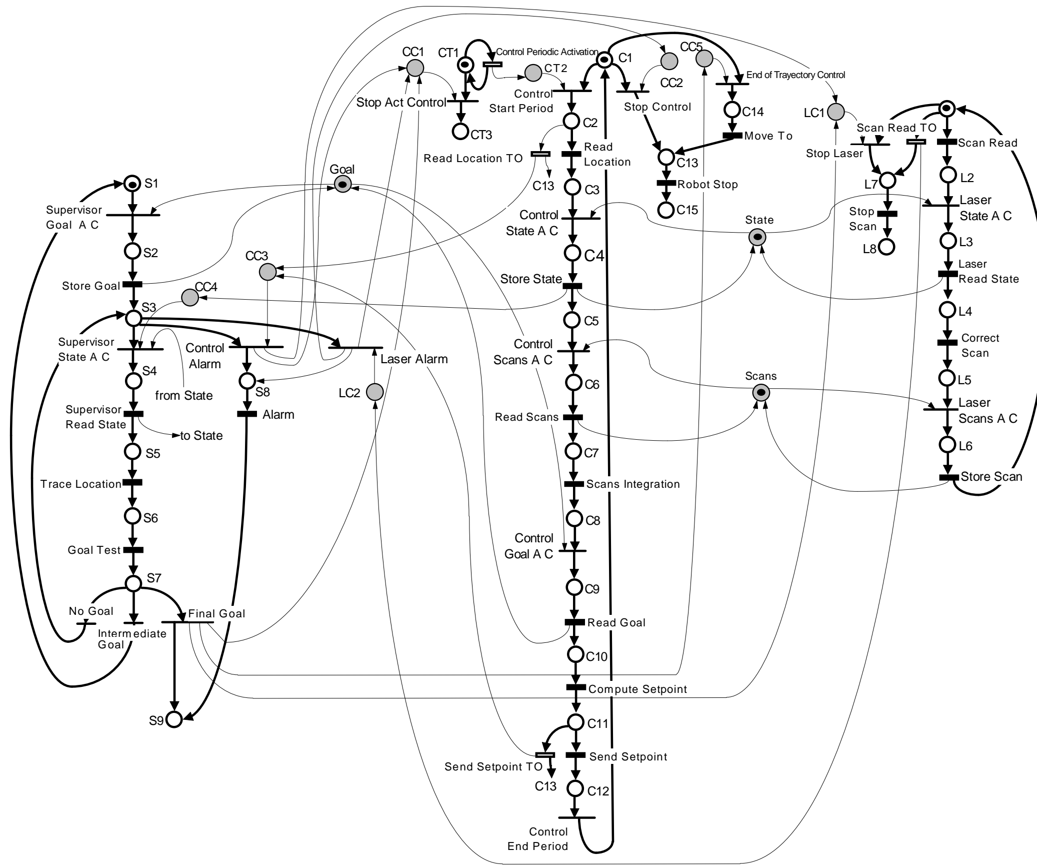

Fig. 4. Sequential processes are highlighted and communication places are shaded in grey

considered in this paper). The priority assignment has been performed using an heuristic method based on RMA techniques.

For the sequential process recognition a p-invariant computation has been developed. There are seven p-invariants: (I_1) Supervisor S_1. S_9, (I_2) Control $C_1..C_{15}$, (I_3) Activation CT_1, CT_3, (I_4) Laser $L_1..L_8$, (I_5) Protected Goal Goal, S_2, C_9, (I_6) Protected State State, L_3, C_4, S_4, (I_7) Protected Scans Scans, L_6, C_6. With these p-invariants the transition coverability problem can be solved, see fig 5. In this table it can be seen that there are four essential p-invariants (I_1, I_2, I_3, I_4), which cover all the net transitions (shaded in grey). The first set of transitions in ME (covered by I_1) corresponds to *Supervisor* process. The second (covered by I_2) corresponds to the *Control* process. The third (I_3), is the periodical activator of the Control process, the *Control_Activation* process, and the fourth (I_4) is the process which deals with the laser, the *Laser* process. There are several places remaining which do not belong to any process, which are modeling asynchronous communications between the processes. The places are CC_4, CC_3, LC_2, CC_1, CT_2, LC_1, CC_2, CC_5, all of private destination, and *Goal*, *State*, *Scans*, of non

	I_1	I_2	I_3	I_4	I_5	I_6	I_7
Store_Goal	x				x		
Supervisor_Read_State	x					x	
Trace_Location	x						
Goal_Test	x						
Alarm	x						
Supervisor_Goal_Access_Control	x				x		
Supervisor_State_Access_Control	x					x	
No_Goal	x						
Intermediate_Goal	x						
Final_Goal	x						
Control_Alarm	x						
Laser_Alarm	x						
Move_To		x					
Robot_Stop		x					
Read_Location		x					
Store_State		x				x	
Read_Scans		x					x
Scans_Integration		x					
Read_Goal		x			x		
Compute_Setpoint		x					
Send_Setpoint		x					
Control_Scans_Access_Control		x					x
Control_Goal_Access_Control		x			x		
Control_State_Access_Control		x				x	
Control_Start_Period		x					
Stop_Control		x					
Control_End_Period		x					
End_of_trajectory_Control		x					
Read_Location_TO		x					
Send_Setpoint_TO		x					
Stop_Act_Control			x				
Control_Periodic_Activation			x				
Scan_Read				x			
Laser_Read_State				x		x	
Correct_Scan				x			
Store_Scan				x			x
Stop_Scan				x			
Laser_State_Access_Control				x		x	
Laser_Scans_Access_Control				x			x
Stop_Laser				x			
Scan_Read_TO				x			

Fig. 5. Covering table of transitions

private destination. E.g. the code corresponding to Control process is shown:

```
task body Control is
 Read_Location_TO: constant duration:=0.1;
 Send_Set_Point_TO:constant duration:=0.1;
begin
  loop
    select
      accept Control_Star_Period;
      select
        delay Read_Location_TO;
        CC3.Mark;    exit;
      then abort
        Read_Location;
      end select;
      State.Demark;  Store_State;
      State.Mark;    CC4.Mark;
      Scans.Demark;  Read_Scans;
      Scans.Mark;    Scans_Integration;
      Goal.Demark;   Read_Goal;
      Goal.Mark;     Compute_Setpoint;
      select
        delay Send_Set_Point_TO;
        CC3.Mark;    exit;
      then abort
        Send_Setpoint;
      end select;
    or
      accept Stop_Control;
      exit;
    or
      accept End_Of_Trajectory;
      Move_To;       exit;
    end select;
  end loop;
  Stop_Robot;
end;
```

6. CONCLUSIONS AND FUTURE WORK

Time Petri nets have been proposed as the formalism for the whole life cycle of real-time systems providing the following advantages: it allows an unambiguous and easy to understand system specification due to its graphical nature; it allows the verification and validation of the correction of the system in the early stages of the cycle; it allows a high modeling flexibility, since it will no longer be necessary to impose restrictions on the system in order to analyze the temporal behaviour and verify the timing constraints. Structural techniques have been applied in order to detect the concurrent sequential processes embedded in the net, avoiding the problems of centralized techniques. Moreover, the implementation is automatizable allowing us the automatic code generation, preventing us from making mistakes during the codification and simplifying the development of the system.

For the computation of sequential processes we have used the net without time information. But sometimes, due to time interpretation, a set of transitions which are not in ME in the underlying Petri net, are in the time Petri net. In future works we will try to detect and formalize this temporal mutual exclusion A further line of research will be the study of the shedulability of systems modeled with TPN since, up to now, it has been performed using heuristic rules. The appropriate priority assignment policy for a process consisting of a set of transitions of different priority must be studied: a static priority equal to the highest of the transition priority in the set, a dynamic priority depending on the transition which is currently fired, or an alternative priority assignment.

ACKNOWLEDGEMENTS

This work has been supported in part by project TAP97-0992-C02-01 from the CICYT of Spain.

References

Aalst, W. v. d. and Odijk, M. A. (1995). Analysis of railway stations by means of interval timed coloured petri nets. *Real-Time Systems*, 9(3):241–263.

Berthomieu, B. and Diaz, M. (1991). Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. on Soft. Eng.*, 17(3):259–273.

Buy, U. and Sloan, R. (1994). Analysis of real-time programs with simple time petri nets. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 228–239.

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

García, F. J. and Villarroel, J. L. (1996). Modelling and ada implementation of real-time systems using time petri nets. In *Proc. 21st IFAC/IFIP Workshop on Real-Time Programming*, Gramado - RS, Brazil.

Gedela, R. and Shatz, S. (1997). Modelling of advanced tasking in ada-95: A Petri net perspective. In *Proc. 2nd Int. Workshop on Soft. Eng. for Parallel and Distributed Systems*, Boston, USA.

Shatz, S. M., Tu, S., Murata, T., and Duri, S. (1996). An application of petri net reduction for ada tasking deadlock analysis. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1307–1322.

Villarroel, J. L. (1990). *Integración Informática del Control en Sistemas Flexibles de Fabricación.* PhD thesis, Universidad de Zaragoza, María de Luna 3 E-50015 Zaragoza, España.