Translating Time Petri Net Structures into Ada 95 Statements *

F.J. García[1] and J.L. Villarroel[2]

[1] Universidad de La Rioja, Dpto. Matemáticas y Computación,
C/ Luis de Ulloa s.n., 26004 Logroño, Spain
fgarcia@dmc.unirioja.es
[2] CPS, Universidad de Zaragoza, Dpto. de Informática e Ing. de Sistemas,
C/ María de Luna 3, 50015 Zaragoza, Spain
JLVillarroel@mcps.unizar.es

Abstract. The intention of this paper is to show how real-time systems modeled with time Petri nets can be implemented in Ada 95. To achieve this objective, we use models of the Ada 95 tasking statements. Using reduction rules the model of the statement is reduced in order to make it recognizable in the net which models the system. Thus, we can build a catalogue of the reduced models of the Ada 95 tasking statements so that they can be used in the translation of net structures into Ada programs.

1 Introduction

This work is a part of a wider study whose main aim is the use of a formal method for the whole life cycle of a real-time system (RTS) development. The complexity of the design, analysis and implementation of real-time systems is well known; a complexity that is further amplified if reliability aspects are considered. In these systems, reliability is essential due to the possible catastrophic effects that a failure could produce. Therefore, any attempt to apply a formal method to the life cycle of an RTS must be welcome so that the reliability requirement can be met. If, in addition, the use of a automated tool to generate the code for the system is considered, this is a double advantage, because in addition to the obvious reduction in the coding mistakes, we add a reduction in the cost of the development. Bearing this idea in mind, our proposal is to use the *Time Petri Net* (TPN) formalism to model, analyze and generate the code for an RTS. In this paper we are concerned with the latter part of this development, that is, the automatizable code generation for an RTS. Ada 95 was chosen as the target language because of its tasking and real-time features.

Petri nets are suitable for the modeling of real-time systems because the net can naturally model concurrency, resource sharing with mutual exclusion, synchronizations, etc. In order to model an RTS with a Petri net, some kind of extension to classical Petri nets involving time must be used. There seems to be some consensus in using the Time Petri Net formalism [1][2] among the

* This work has been partially supported by the CICYT (project TAP97-0992-C02-01)

authors who have dealt with this subject in the recent past. See for example [3][4][5], or [6][7] where only classical nets are used. Nevertheless, other related formalisms, such as the interval timed coloured Petri nets [8] or the time environment/relationship nets (TER nets) [9] must not be dismissed, above all the latter in the modeling of absolute delay statements. We have chosen time Petri nets due to their high expressive power (with the only limitation being the modeling of absolute delay) 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.

The idea of bringing together the worlds of Ada and the Petri nets is not new. Several authors have already used Petri nets for the modeling of Ada and Ada 95 tasking statements, mainly with two objectives: to define precise behaviour for tasking semantics and to support the automated analysis of concurrent software, above all in deadlock detection. For the first objective, see for example [3][4] where formal models for the main tasking statements are shown. The second line of interest is the subject matter in [5][6][7].

The intention of this paper is to show how a model of an RTS in terms of time Petri nets can be implemented in Ada 95 and demonstrate that the code generated for this implementation is correct, in the sense that it has the same semantics as the model that is being implemented. To achieve this objective, we use models of the Ada 95 tasking statements, models similar to those in [3][4]. Using reduction rules for classical or time Petri nets ([6][10][11]) we reduce the model to the minimum net structure that can be directly recognized in a time Petri net. Once we have a catalogue of the reduced models of the Ada 95 tasking statements, implementing the software skeleton of an RTS modeled with a time Petri net requires only that we isolate and recognize the reduced models in the net, and put the corresponding Ada statements together in the program that will implement the system. This last part can be automated. In addition to this main objective, this work can be seen as a contribution in two additional ways: it complements the definition of the models of the Ada 95 statements, as in [3][4], and it helps with the proposal described in [6] since it provides reduced models of the tasking structures that can reduce the size of the model of the system to be analyzed.

The organization of this paper is as follows. Section 2 summarizes basic concepts about time Petri nets and their relation with real-time systems. In section 3, we describe how a time Petri net can be decomposed into a set of concurrent processes, each of which will be implemented in a different Ada task. Section 4 shows an example of implementation of a real-time system modeled with a TPN and the next section explains how the code of the example was built.

2 Basic Concepts about Time Petri Nets

A Time Petri Net ([2]) is a tuple $(P, T; F, B, M_o, SIM)$, where $(P, T; F, B, M_o)$ defines a marked classical Petri net, the *underlying Petri net*[1]; and SIM is the

[1] We assume a basic knowledge about classical Petri nets (see [10] for a survey)

mapping called the static interval $SIM : T \rightarrow \mathbb{Q}^* \times (\mathbb{Q}^* \cup \infty)$, 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) which are associated with transitions. Assuming that transition t_i is enabled at time θ_0, and is 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 remain enabled without firing. These two time values therefore 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. This work assumes that a transition with no associated time interval has an implicit time interval of (0, 0), that is, the transition is immediately fired as soon as it becomes enabled. The need to implement the net inspired this decision, against the approach taken in [2], where a time interval of (0, ∞) was considered for these transitions. In addition, we use predicates associated with transitions and inhibitor arcs which connect places and transitions in our models of Ada statements (see [10]).

All transitions in TPNs have the same functionality, but the different situations that appear in an RTS must be focused on in our models. Therefore, and with the aim of implementing the model, we distinguish three kinds of transitions:

- CODE transitions (filled in segments) together with their input places, represent the code associated with an activity, which starts its execution when the transition is enabled, i.e. the input places are marked. The two time values (α, β) represent the execution time of the activity. At best, the code execution will finish at time α, and at worst the execution will take until β. The firing represents the end of the code.
- TIME transitions, (unfilled segments) are those with an associated time event, e.g. a time-out. They also have associated time information, described with an interval (α, α), where α represents the event time. The firing of this kind of transition represents the occurrence of the event.
- SYCO transitions (thin segments) are those with no temporal meaning used to perform synchronization (SY) and control (CO) tasks. The firing of a SYCO transition leads to simple state changes.

As an example, Fig.1 shows a TPN modeling a periodic process that executes a piece of code and communicates with another process. 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 A models the periodic activation of the process. Every 10 time units, the transition fires and causes the execution of the process. Box B shows an action, i.e. code, to be executed by the process. The execution starts when the input place is marked. The computation time of this activity is between 4 and 5 time units. 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 (start the communication) before $\tau + 1$ (expiration time of the time-out), then transition (1, 1) will fire, aborting the starting of the communication.

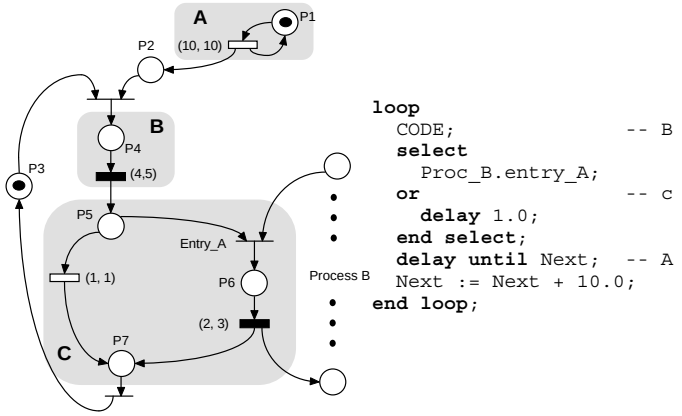

Fig. 1. Example of TPN model

3 Decomposing Petri Nets into Processes

A time Petri net can describe the behaviour of an RTS. Several concurrent activities or processes can usually be recognized in this behaviour. Thus, the first step in the implementation of an RTS modeled with a time Petri net is to isolate these processes and their inter-connections. A brief description about how this decomposition can be achieved is given here (see the details in [12][13][14][15]). The basis is to merge a set of transitions in mutual exclusion (ME) into a single process (two transitions are in mutual exclusion if they cannot be fired simultaneously). A set of transitions which are in ME are not concurrent, so guaranteeing that a process is made up only of sequential activities. The existence of ME between transitions can be determined by a computation of *monomarked p-invariants*[2] because they describe a set of places in ME which obviously implies a set of transitions in ME (the input and output transitions of the places). In addition, the p-invariant can be used to describe the control flow of the process. In this way, the only transitions which are able to fire are those whose input place belonging to the p-invariant is marked (see for example Fig. 1, where a token passes through places p_3, p_4, p_5, p_6 and p_7, determining the flow of the process).

Eventually some places will remain which do not belong to any process. These places model asynchronous communications between the processes which they link. The other way of communication is a shared transition which represents a synchronous communication (rendezvous). Moreover, it is possible to share

[2] A monomarked p-invariant is a set of places interconnected with transitions through which a single token flows (monomarked) and that holds that in every reachable state the token is always in one of these places, i.e., the sum of the tokens in the places is always one (invariant)

sets of transitions and places grouped in a subnet; this subnet represents the execution of a piece of code in an extended rendezvous.

4 An example

Let us consider the example of Fig. 2 in which we show the model of an RTS made up of a periodical activity (Activity 1) of period *Period_Activation* that executes a code (*code 1*) which has an associated time-out (*time-out 1*). After this execution, a rendezvous with another activity takes place. This rendezvous has another associated time-out (*time-out 2*). Activity 2 executes a code (*code 3*) that can be aborted if the time-out of *code 1* is fired. Otherwise, Activity 2 makes a rendezvous with Activity 1 and later executes a new piece of code (*code 4*). The whole system stops if *time-out 1* is fired.

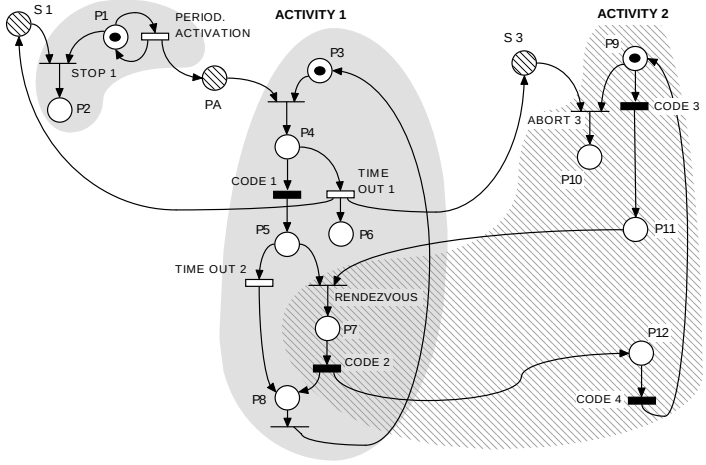

Fig. 2. A real-time system modeled with a TPN. The processes are highlighted

A computation of monomarked p-invariants reveals the existence of three of them which cover all the transitions of the net. They are: $I_1 = \{p_1, p_2\}$; $I_2 = \{p_3, p_4, p_5, p_6, p_7, p_8\}$; $I_3 = \{p_7, p_9, p_{10}, p_{11}, p_{12}\}$. With these p-invariants, three processes can be built (they are highlighted in Fig. 2). One of them (*Proc_1*) represents the periodical activator of Activity 1, another (*Proc_2*) is the body of Activity 1 and the last (*Proc_3*) corresponds to Activity 2. Fig. 2 also shows the way in which the processes communicate with each other. Three shared places (S_1, PA and S_3) act as a medium for asynchronous communication and one shared subnet ({rendezvous, p_7, code 2}) acts as a synchronous communication medium in which *code 2* is executed.

Due to the fact that in the decomposition technique the temporal information is not taken into account, the periodical activity is split into two processes.

We are currently studying how to include temporal mutual exclusion in the recognition of processes. Observe that if time were considered the transitions of *Proc_1* would all be in mutual exclusion with the ones in *Proc_2*, and the two processes could therefore be brought together.

The implementation will be made up of three Ada tasks and the implementation of the shared places. Here is the Ada 95 code that we propose for the processes and the shared places:

```
task body S1 is
begin
  accept Mark;
  Proc_1.Demark_S1;
end S1;
-- The same structure for
-- the place PA

task body Proc_1 is
  L_E: Time; -- Last_Event
begin
  L_E := CLOCK;
  loop
   select
    accept Demark_S1;
    L_E := CLOCK;
    exit;
   or
    delay until L_E + PERIOD_ACT;
    L_E := L_E + PERIOD_ACT;
    PA.Mark;
   end select;
  end loop;
end Proc_1;

task body Proc_3 is
begin
  loop
    select
      S3.Demark;
      exit;
    then abort
      CODE_3;
    end select;
    Proc_2.Rendezvous;
    CODE_4;
  end loop;
end Proc_3;
```

```
task body Proc_2 is
  L_E: Time; -- Last_Event
begin
  L_E := CLOCK;
  loop
    accept Demark_PA;
    L_E := CLOCK;
    select
      delay until L_E + Time_out_1;
      L_E := L_E + Time_out_1;
      S1.Mark; S3.Mark; exit;
    then abort
      CODE_1;
      L_E := CLOCK;
    end select;
    select
      accept Rendezvous do
        CODE_2;
      end Rendezvous;
      L_E := CLOCK;
    or
      delay until L_E + Time_out_2;
      L_E := L_E + Time_out_2;
    end select;
  end loop;
end Proc_2;

protected body S3 is
  entry Demark when Marks > 1 is
  begin
    Marks := Marks - 1;
  end Demark;
  procedure Mark is
  begin
    Marks := Marks + 1;
  end Mark;
end S3;
```

5 Software implementation

This section describes how the code of the previous example was built. Each process can be implemented in an Ada task that has a loop structure. The flow of the token through the p-invariant that generates the process reveals the execution order of the transitions. Basically the implementation of the transitions are as follows:

- The existence of a SYCO transition means that decisions are taken inside a process or that synchronous communication between processes occurs, as will be shown below.
- The existence of a CODE transition involves the execution of its associated code.
- The existence of a TIME transition represents a delay in the execution of the process. The model of a relative delay statement corresponds to a TIME transition.[3] Therefore, as a first approximation, a TIME transition is implemented with a delay statement. However, the implementation of a TIME transition with a simple delay can provoke cumulative drift in the implemented process. This can be avoided if a time variable is associated with each process containing TIME transitions. This variable (`Last_Event`) records the time at which the last marking update occurred in the process. Each time a transition is fired, the time at which this firing occurs is recorded in the variable. This time is used in the computation of the expiration time of the delays. Thus, the implementation of a TIME transition with an associated time interval (D, D) corresponds to these two instructions:

```
delay until Last_Event + D;
Last_Event := Last_Event + D;
```

The use of this variable can be seen in the code of the example in processes *Proc_1* and *Proc_2*.

The implementation becomes more complex when conflicts appear, i.e. there are several transitions at the output of a place. More complex Ada statements must be used. At this point we begin the proof that the proposed code for the example net is correct, i.e. that it has the same semantics as the model itself. The procedure followed for the proof consist of modeling the semantics of each Ada statement used in the implementation code. By means of a reduction process, we reduce the model of the statement to the structure of the net that was translated into that Ada statement. During the reduction, we must preserve all the CODE and TIME transitions and merge a sequence of SYCO transitions firings to a single transition with the same behaviour, in such a way that the original model and the reduced one are equivalent. Several rules that obey these restrictions

[3] The modeling of a delay until statement is not possible using TPN, because all the time values involved are relative to the instant of the enabling of the transition. To model an absolute delay statement, it would be necessary to consider the use of TER nets [9]

have been taken from [6][10][11] for use in the reduction process. Lack of space prevented us from presenting all the models for every Ada structure. We show some of the more representative models, without considering the occurrence of exceptions, aborts or requeue statements.

5.1 Modeling the select statement with a delay branch

The net in Fig. 3.(1) is proposed to model the behaviour of a select statement that has several accepts and a delay branch. The model corresponds to the situation in which only one client calls the server. The model is not valid if different clients can call all the entries.

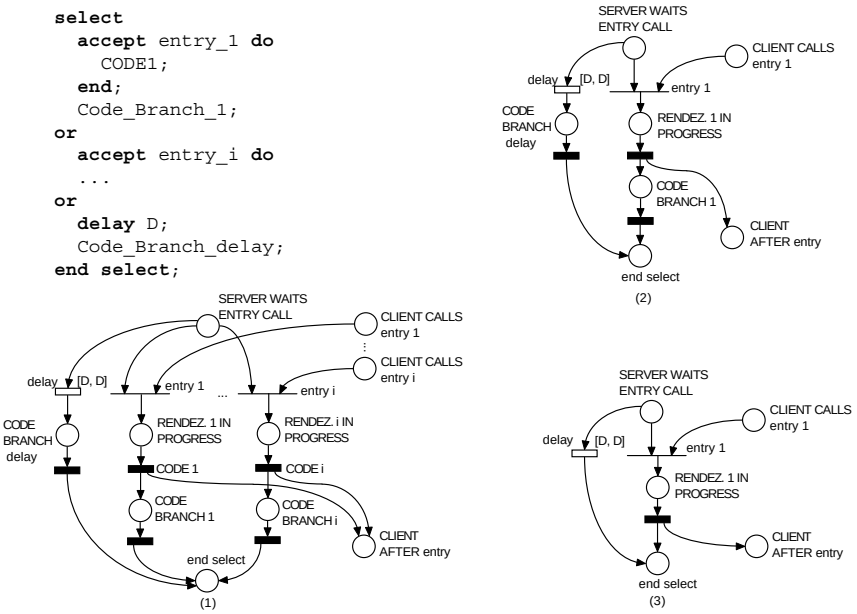

Fig. 3. Model and reduction of the select statement with a delay branch

The server that executes the select statement must wait in the *server waits entry call* place until one of its output SYCO transitions becomes enabled due to a client issuing a call to one of its entries by marking a *Client calls entry i* place. The SYCO transition is then fired, representing the start of the rendezvous in which *code i* is executed. Once the code finishes, the client can continue and the server can execute an optional code (*code branch i*) before finishing the select. If the marking of some *client calls entry i* place does not take place in a time D, the TIME transition corresponding to the *delay* fires, removing the token from the *server waits entry call* place and aborting the possibility of making any rendezvous.

The model can be simplified if we consider only one client and one entry (Fig. 3.(2)), as in the case of our example. If, in addition, the select does not have the optional code after either the accept or the delay branches, then the *code branch delay* and *code branch 1* transitions can be removed (Fig. 3.(3)). This structure can be seen in the example net involving the *rendezvous*, *time-out 2* and *code 2* transitions. This corresponds to the select of *Proc_2* and the entry call of *Proc_3*. Moreover, if no code is executed in the accept, the same structure can be recognized in the *stop 1* and *period. activation* transitions corresponding to *Proc_1*.

With the aim of avoiding cumulative drift, for example in *Proc_1*, we substitute the delay statement which corresponds to the TIME transition with the delay until statement which includes the previously mentioned time variable *Last_Event*. It is immediately obvious that this new implementation has the same behaviour as the one derived from the model.

5.2 Modeling the ATC with a delay triggering alternative

The net in Fig. 4.(1) models the behaviour of an ATC statement with a delay statement as triggering alternative (the model is similar to the one in [4]).

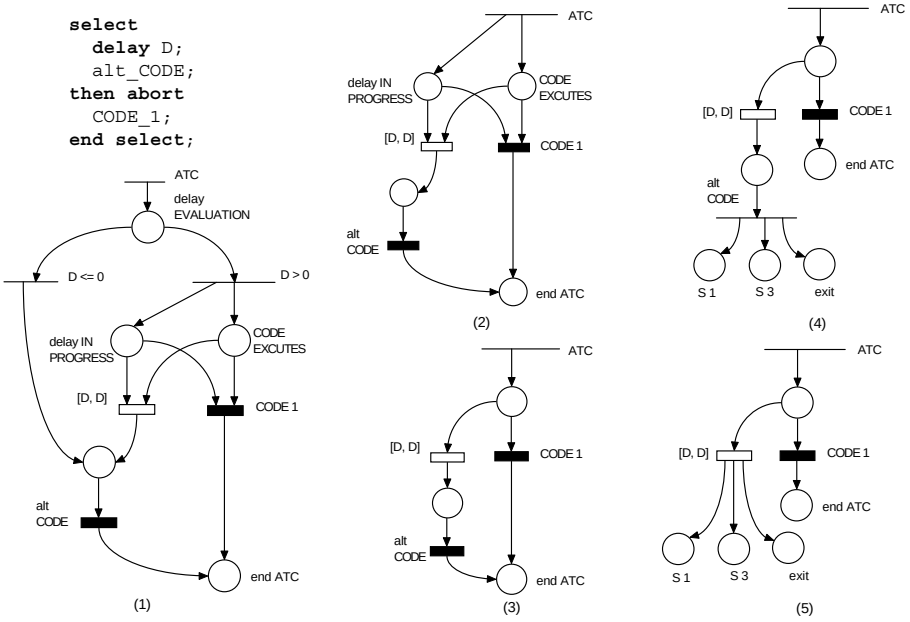

Fig. 4. Model and reduction of the ATC statement with a delay as triggering alternative

Once the ATC begins, the delay expression must be evaluated. If the expression is positive, the *delay in progress* and *code executes* places are marked,

involving the enabling of the TIME and CODE transitions. These two transitions compete to remove the tokens from both places. If the CODE transition *code 1* fires before time D expires, the tokens will be removed and the TIME transition will be disabled (this means aborting the delay statement). The code of the delay alternative is then executed. On the other hand, if time D expires before the CODE transition fires, the tokens are removed and the CODE transition disabled (this means that the code execution is aborted). If the expression in the delay is negative or zero, the code of the delay alternative is directly executed. This case has not been considered in the reduction process because, within the scope of this study, the delay expression is always positive (Fig. 4.(2)).

Applying reduction rule 6 of [11] (parallel redundant places) the *delay in progress* and *code executes* places can be reduced to one (Fig. 4.(3)). In the case of our example, the code that must be performed if the delay alternative is triggered is the marking of two places (S_1 and S_3) and the execution of an exit statement that breaks the normal flow of the process, avoiding the place *end ATC* being marked. This is shown in Fig. 4.(4). In Fig. 4.(5) the final structure is obtained by applying rule 3 of [11] (post-fusion). The resultant model can be recognized in the net of the example involving place p_4 and transitions *code 1* and *time-out 1* that are implemented in *Proc_2*. Once again, the delay is substituted with the corresponding delay until using the *Last_Event* variable.

5.3 Modeling the ATC with an entry call triggering alternative

The net in Fig. 5.(1) models the general behaviour of an ATC statement with an entry call as triggering alternative. This model corresponds to the general real situation, where both the triggering alternative and the abortable part can evolve in parallel. However, we consider only monoprocessor implementation platforms, and static priorities. With these restrictions, the model can be simplified because it is impossible for both parts to be executed at the same time. In this case, the server that accepts the entry call must have a higher priority than the client, since this is the only way in which the server can have a chance of interrupting the abortable part. Once the entry is accepted, the abortable part is preempted and the ATC finishes before it can execute again. This simplification leads us to the model in Fig. 5.(2).

When the ATC begins, *abortable part* and *wait for accept* places are marked. This involves *code 1* starting its execution while the marking of the *server accept* place is being waited for. If this is already available, the transition *rendezvous* fires, removing the token from the *abortable part* place and aborting the starting of the code. Otherwise, the code execution will be aborted in the same way at the moment in which the *server accept* place is marked. If this place is not marked, the code can finish removing the token from the *wait for accept* place and then abort the entry call.

For the reduction rule 2 of [6] is used (pre-fusion of transitions). This leads us to the net in Fig. 5.(3). Rule 6 of [11] (parallel redundant places) can then be used to eliminate the *wait for accept* place (which is the same as *abortable part*). The code that is executed after the entry call consists of an exit statement

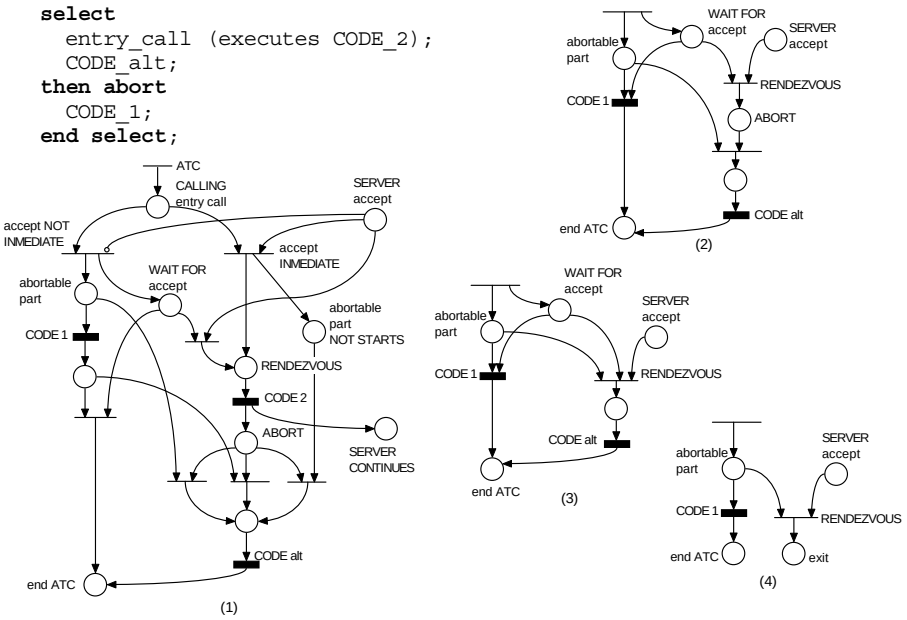

Fig. 5. Model and reduction of the ATC statement with an entry call as triggering alternative

that breaks the normal flow of the process and avoids the *end ATC* place being marked. The product of these last steps is shown in Fig. 5.(4). This structure can be directly seen in the net of the example involving p_9, S_3 and p_{10} places, and *abort 3* and *code 3* transitions.

6 Conclusions and Future Work

The Petri net formalism is directly executable. This paper demostrates how the code which implements the net can be obtained. Moreover it has been shown that the translation into Ada 95 code is correct, i.e. it has the same behaviour as the net. This allows us to enrich the Petri nets formalism, which has been traditionally used for the specification and analysis of behavioural and timing properties in real-time systems. We have now shown how to implement this, and so making time Petri nets an alternative method for the whole life cycle of a real-time system.

Future work will be devoted to the study of several optimizations for the generated code. Firstly a study must be carried out on how to include temporal restrictions in the recognition of the processes embedded in the time Petri net. This will avoid situations such as the one in the example of section 4, where a periodic activity had to be split into two processes. In the second place, the use of the time variable *Last_Event* in order to avoid cumulative drift must be

optimized. In the current implementations it is used in every process with TIME transitions but should anly be necessary in process where there is at least one sequence of execution only with TIME transitions and no external interaction.

References

[1] P. Merlin and D.J. Farber. Recoverability of communication protocols. *IEEE transactions on Communication*, 24(9), September 1976.

[2] B. Berthomieu and M. Diaz. Modeling and verification of time dependent systems using time Petri nets. *IEEE transactions on Software Engineering*, 17(3):259–273, March 1991.

[3] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato. Modeling the Ada task system by Petri nets. *Computer Languages*, 10(1):43–61, 1985.

[4] R.K. Gedela and S.M. Shatz. Modelling of advanced tasking in Ada-95: A Petri net perspective. In *Proc. 2nd Int. Workshop on Software Engineering for Parallel and Distributed Systems, PSDE'97*, Boston, USA, 1997.

[5] U. Buy and R.H. Sloan. Analysis of real-time programs with simple time Petri nets. In *Proc. Int. Symp. on Software Testing and Analysis*, pages 228–239, 1994.

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

[7] S. Duri, U. Buy, R. Devarapalli, and S.M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada. *ACM Transaction on Software Engeeniering Methodology*, 3(4):340–380, December 1994.

[8] W.M.P. van der Aalst and M.A. Odijk. Analysis of railway stations by means of interval timed coloured Petri nets. *Real-Time Systems*, 9(3):241–263, November 1995.

[9] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A unified high-level Petri net formalism for time-critical systems. *IEEE transactions on Software Engineering*, 17(2):160–171, February 1991.

[10] T. Murata. Petri nets: properties, analysis, and applications. *Proceedings of the IEEE*, 77(4), April 1989.

[11] R.H. Sloan and U. Buy. Reduction rules for time Petri nets. *Acta Informatica*, 43:687–706, 1996.

[12] J.M. Colom, M. Silva, and J..L. Villarroel. 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, January 1986.

[13] F. Kordon. Proposal for a Generic Prototyping Approach. In *IEEE Symposium on Emerging Technologies and Factory Automation, Tokyo, Japan*, number 94TH8000, pages 396–403. IEEE Comp Soc Press, 1994.

[14] F. Bréant and J.F. Peyre. An improved massively parallel implementation of colored Petri nets specifications. In *IFIP-WG 10.3 working conference on programming environments for massively parallel distributed systems, Ascona, Switzerland*, 1994.

[15] F.J. García and J.L. Villarroel. Decentralized implementation of real-time systems using time Petri nets. application to mobile robot control. In D.F. García Nocetti, editor, *Proc. of the 5th IFAC/IFIP Workshop*, Algorithms and Architectures for Real Time Control 1998, pages 11–16. Pergamon, 1998.