Schedulability Analysis of Pre-runtime and Runtime Scheduling Algorithm of an Industrial Real Time System

Stefano Pepi(✉) and Alessandro Fantechi(✉)

DINFO, University of Florence, Via S. Marta 3, Florence, Italy
{stefano.pepi,alessandro.fantechi}@unifi.it

Abstract. The configuration of a complex, generic, real-time application into a specifically customized signalling embedded application has an important impact on time to market, deployment costs and safety guarantees for a railway signalling manufacturer. In this paper we focus on the aspect of real-time schedulability analysis, that takes an important portion of the time dedicated to configuration in this kind of systems. We propose an approach based on rigorous modelling of the scheduling algorithms, aimed at substituting possibly unreliable and costly empirical tuning. In order to comply with the needs of our industrial partners, we have resorted to the use of variants of Petri Nets with associated available tools: Timed Petri Nets (TPN) and Coloured Petri Nets (CPN), supported by open source tools, respectively *TINA* and *CPN Tools 4.0* have been exploited for the modelling of the pre-runtime and the runtime scheduling algorithms implemented in the industrial platform. The comparison of models produced with the two tools has concluded that the Coloured Petri Nets are more suited to the adopted schedulability analysis approach, for both scheduling algorithms.

Keywords: Petri Nets · Timed Petri Nets · Coloured Petri Nets · Real Time Systems · Scheduling algorithm · Modelling · Formal verification · Railway signalling

1 Introduction

Real-Time Systems (RTS) are those computer-based systems where correct operation does not only depend on the correctness of the results obtained, but also on the time at which the results are produced [21].

The interest for real-time systems is motivated by many applications that require that computations satisfy given time constraints, in domains such as automotive, avionics, communications, railway signalling etc.

The most important property of a RTS is *predictability*. Predictability is the ability to determine in advance if the computation will be completed within the time constraints required. Predictability depends on several factors, ranging from the architectural characteristics of the physical machine, to the mechanisms

S. Hammoudi et al. (Eds): MODELSWARD 2016, CCIS 692, pp. 46–69, 2017.
DOI: 10.1007/978-3-319-66302-9_3

of the core, up to the programming language. Predictability can be measured as the percentage of processes for which the constrains are guaranteed.

In this article we report the experience made in collaboration with our industrial partner, a railway signalling manufacturing company, in the implementation of a generic real-time platform based on a proprietary microkernel Real Time Operating System; in particular we present a method for schedulability analysis.

With the recent expansion of markets to Asia and Africa, the company has experienced a growing need for a versatile system that can be configurable for each different application. The transition from a traditional "main loop"-based system to a general purpose platform has allowed low-cost configuration, simply by changing the application inside and the hardware to interact with. With the same Hw/Sw platform both *ground* and *on-board* systems can be built, either for urban (like metro) or main line applications, meeting the signalling regulations of different countries.

Experience has however shown that guaranteeing predictability for the different customizations of the platform takes a considerable portion of the customization effort, if based only on testing every time the newly customized software on the platform.

We have therefore considered the possibility of building a generic model of the scheduling algorithms employed in the platform, that is going to be instantiated on the temporal constraints and tasks numbers of the different specific applications (that is, customizations), in order to support the validation of predictability by means of proper model simulation tools.

Basing on the wide literature about modelling real-time systems with Petri Nets (see, for example, [3,5,10,11]) and on the availability of related tools, we have chosen to experiment two Petri Nets dialects for the modelling of the scheduling algorithms, in order to predict schedulability of the set of tasks governing a new specific application. Both Timed Petri Nets (TPN) and Coloured Petri Nets (CPN) have been evaluated for this purpose, together with their support tools, favouring at the end the adoption of Coloured Petri Nets.

Due to the limited time available to conduct the experiments, in order to satisfy stringent temporal requirements from our industrial partner, we have chosen not to investigate other temporal modelling formalisms, such as timed automata [2]. The results obtained by these experiments were however judged sufficiently satisfactory to consider the adoption of the technique inside the development process of our industrial partner.

This paper is structured as follows: the next section introduces the industrial context that has motivated our work on modelling scheduling algorithms; in Sect. 3 we present the background of the modelling method, namely the two considered variants of Petri Nets, while in the next two sections we present the models of the pre-runtime and runtime scheduling policies. Section 6 compares the models obtained with the two Petri Nets variants, and Sect. 7 draws some conclusions.

2 Scheduling in Safety-Related RT Applications

A real-time process is characterized by a fixed time limit, which is called *deadline*. A result produced after its deadline is not only late, but can be harmful to the environment in which the system operates. Depending on the consequences of a missed deadline, real-time processes are divided into two types:

- *Soft real-time*: if producing the results after its deadline has still some utility for the system, although causing a performance degradation, that is, the violation of the deadline does not affect the proper functioning of the system;
- *Hard real-time*: if producing the results after its deadline may cause catastrophic consequences on the system under control.

To meet real-time requirements, scheduling plays an important role. Depending on the assumption done on the processes and on the type of hardware architecture that supports the application, the scheduling algorithms for real-time systems can be classified according to the following orthogonal characteristics:

- *Uniprocessor* vs. *Multiprocessor*
- *Preemptive* vs. *No preemptive*
- *Static* vs. *Dynamic*
- *Pre-runtime (offline)* vs. *Runtime (online)*
- *Best-Effort* vs. *Guaranteed*

For what concerns the fourth characteristic, in pre-runtime scheduling all decisions are taken before the process activation on the basis of information known a priori. The schedule is stored in a table which will be integrated into a run-time kernel. The kernel has one component called *dispatcher* which takes tasks from the table and loads them onto the processing elements, according to specified timing constraints. The *Runtime* category represents instead those algorithms in which the scheduling decisions are made at runtime on all currently active processes. The ordering of tasks is then recalculated for each new activation.

In our case the platform is able to manage both these two types of scheduling. In fact according to the type of field application it is possible to enable one or the other algorithm. The choice is made based on the level of safety that the system must ensure.

CENELEC EN50128 is the standard that specifies the procedures and the technical requirements for the development of programmable electronic devices to be used in railway control and signalling protection [7]. This standard is part of a family, and it refers only to the software components and to their interaction with the whole system. The basic concept of the standard is the *SIL* (Safety Integrity Level). Integrity levels characterize software modules and functions according to their criticality, and range is defined from 0 to 4, where 0 is the lowest level, which refers to software functions for which a failure has no safety effects and 4 is the maximum level, for which a software failure can have severe effects on the safety of system, resulting in possible loss of human life.

The pre-runtime scheduling algorithm is used for those application that are classified at SIL 4, since it gives the possibility to fully demonstrate predictability, which is a must in a safety-critical environment. Indeed, with pre-runtime scheduling it is possible to exhibit to an assessor the analyses conducted on the considered set of tasks in order to establish that tasks, with the a priori fixed execution order, do not miss their deadlines. On the other hand, with run-time scheduling algorithms, evidences provided simply by running tests can be not convincing about their coverage of all possible cases, due to possible different run-time scheduling choices. For this reason the run-time scheduling is used for applications of lower SIL.

Indeed, the present paper aims to show a method to strengthen the analysis on pre-runtime and runtime scheduling to a high level of confidence. In particular, we present a method that can be used to verify the pre-runtime schedulability of a task set that contains only periodic tasks with time and priority constrains. The method can be used also to simulate the behaviour of a runtime scheduler with a given taskset, in order to improve confidence on their run-time schedulability.

The motivations for this approach come also from the high variability of installations of the same signalling system at different locations or controlling different stations or lines. Indeed, in railway signalling systems, a distinction is often done between *generic applications* and *specific applications* (as in the already cited CENELEC EN50128 [7] guidelines): generic software is software which can be used for a variety of installations purely by the provision of application-specific data and/or algorithms. A specific application is defined as a generic application plus configuration data, or plus specific algorithms, that instantiate the generic application for a specific purpose.

While the platform is part of a generic application, and hence it is validated once for all, for each specific application the satisfaction of real-time constraints must be verified from scratch.

Indeed, quite often in everyday work it is necessary to revise the schedule of some systems, and all this is routinely done in an empirical way. It is clear that each specific application has a different way to interact with the platform and especially with its resources, such as, for example, input/output drivers for different hardware. It is for this reason that the schedule of real-time tasks should be revised at any new specific application.

The adopted empirical approach includes actions to be taken when configuring the platform for a new specific application, such as: get a new schedule configuration offline and test it on the target. It rarely happens that the first test is successful.

The estimated effort required for the identification and testing of a new scheduling configuration can be summarized with the following parameters:

- **Offline Identification Time:** time needed in order to design the new schedule, it is usually about 30 min (not necessary for runtime scheduling).
- **Flashing Time:** the time needed to load the scheduling on the target, 15 min.
- **Startup Time:** start-up time of the platform, 1.5 min.

- **Running Time:** time during which the system must run without exhibiting timing problems, 30 min/1 h.
- **Attempts:** average number of attempts to get the scheduling, 3.

Summing all the times shown above we get that for each test scheduling, the whole process easily reaches 8 h, which means an entire working day. This process can be automated by a tool that, given a task set and a number of constraints, is able to produce a feasible scheduling. This would mean a huge saving in terms of man hours used to refine the scheduling. Moreover, an empirical evaluation of schedulability of a given dataset does not guarantee that the deadlines are met in any case, putting in danger the overall safety of the system. Using a rigorous approach to the analysis of the schedulability will improve hence the conformance, of a specific application, to safety guidelines.

3 Proposed Method

The rigorous approach we propose is based on the use of Petri Nets to build a model of the scheduling algorithm. A Petri Net [17–19] is a mathematical representation of a distributed discrete system. As a modelling language, it describes the structure of a distributed system as a bipartite graph with annotations. A Petri Net consists of places, transitions and directed arcs. There may be arcs between places and transitions but not between places and places or transitions and transitions.

The places can hold a certain number of tokens and the distribution of tokens on all the places of the network it's named marking. Transitions act on input tokens according to a rule, that is named *firing rule*.

A transition is enabled if you can fire it, that is, if there are tokens in every input place. When a transition fires, it consumes tokens from its input places and places a token in each of its output places.

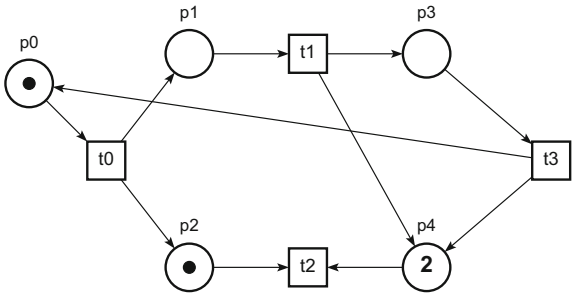

Fig. 1. Representation of an ordinary Petri Net.

Figure 1 shows an example of an ordinary Petri Net. The execution of Petri Nets is nondeterministic, that is, if there are more transitions enabled at the same time any of them can fire. Since taking a transition is not predictable in advance, Petri Nets are well suited for modelling the concurrent behavior of distributed systems.

Formally we can define a Petri Net as a tuple $PN = (P, T, F, W, M_0)$ where:

- P is a finite set of *places*;
- T is a finite set of *transition*;
- $F \subseteq (PxT) \cup (TxP)$ is a set of *arcs*;
- $W : F \to \mathbb{N}$ represents the weight of the flow relation F.
- $M_0 : P \to \mathbb{N}$ is the initial marking vector, which represents the initial state of system.
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

3.1 TPN

A Timed Petri Net is a Petri Net extended with time. In Timed Petri Nets, the transitions fire in "real-time", i.e., there is a (deterministic or random) firing time associated with each transition, the tokens are removed from input places at the beginning of firing, and are deposited into output places when the firing terminates. Formally we can define a Timed Petri Net [20] as a tuple $TPN = (PN, I)$ where:

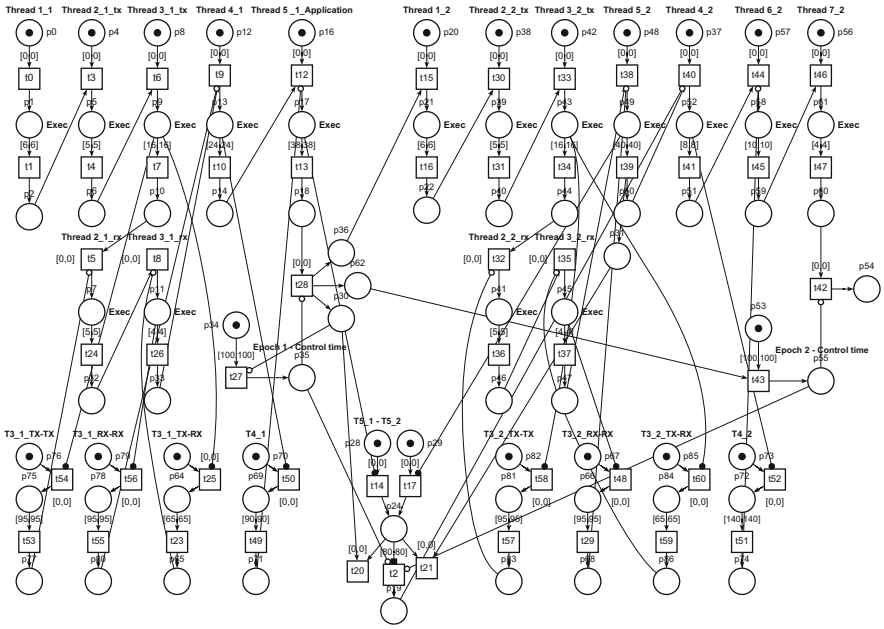

Fig. 2. Timed Petri Net model for a fixed scheduler.

- PN is a standard Petri Net;
- $I : T \to \mathbb{N} \times \mathbb{N}$ is a function that maps each transition to a bounded static interval
- $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

3.2 CPN

An ordinary PN has no types and no modules, only one kind of tokens and the net is flat. With *Coloured Petri Nets* (CPNs) it is possible, instead, to use data types and complex data manipulation. In fact each token has attached a data value called the *token colour* of a given data type: the type defines the range of values that the attributes can assume and the operations applicable in the same way of a variable type in any programming language. The types can be basic types or structured types, the latter defined by the user. The token colour values can be inspected and modified by the occurring transitions.

Formally we can define a Coloured Petri Net as a tuple $CPN = (P, T, F, \Sigma, C, N, E, G, I)$ where:

- P is a finite set of *places*;
- T is a finite set of *transition*;
- $F \subseteq (PxT) \cup (TxP)$ is a set of *arcs*;
- Σ is a set of data types (colour domains).
- C is a colour function. It maps places in P into colours in Σ.
- N is a node function. It maps A into $(PxT) \cup (TxP)$.
- E is an arc expression function. It maps each arc $a \in A$ into an expression e with values in Σ. The input and output types of the arc expressions must correspond to the type of the nodes the arc is connected to.
  The node function and the arc expression function allows multiple arcs to connect the same pair of nodes with different arc expressions.
- G is a guard function. It maps each transition $t \in T$ into a guard expression g, evaluated to a boolean value.
- I is an initialization function. It maps each place $p \in P$ into an initialization expression i. The initialization expression must evaluate to a multiset of tokens with a colour corresponding to the colour $C(p)$ of the place p.

With CPNs it is possible to build a hierarchical description, so that a large model can be easily obtained by combining a set of submodels.

4 Modelling the Pre-runtime Scheduling

We provide now the taskset and the constraints for the fixed scheduler, and then the related models, expressed in the two variants of Petri Nets.

4.1 Taskset and Constraints Specification

In our system the application is decomposed into a set of tasks $\tau_i : i = 1, ...n$ and for this paper we only consider periodic tasks, and we assume that non-periodic tasks are carried out by a periodic server, or processed in the background [6]. The temporal model mostly used in real-time scheduling theory is an extension of the model of Liu and Layland [16] where each task τ_i is characterized by the following parameters:

- R_i: first release time of τ_i;
- C_i: run time of τ_i, which is its worst case execution time (WCET);
- D_i: relative deadline of τ_i, the maximum time elapsed between the release of an instance of τ_i and its completion;
- P_i: release period of τ_i.

In the following we use as a running example the case of a real signalling application, an interlocking system. An *interlocking* system is the safety-critical system that controls the movement of trains in a station and between adjacent stations. The interlocking monitors the status of the objects in the railway yard (e.g., points, switches, track circuits) and allows or denies the routing of trains in accordance with the railway safety and operational regulations that are generic for the region or country where the interlocking is located. The instantiation of these rules on a station topology is stored in the part of the system named control table that is specific for the station where the system resides. We refer to [9] for a review on the vast literature on formal modelling of interlocking systems. In this context, we are interested instead to focus on the characteristics of the task set of this application, consisting of 7 threads which have the following goal:

- T_1 is in charge of operating on the Ethernet channel;
- T_2 is one of the most important thread and it is in charge of the safety of the system;
- T_3 implements a protocol stack for the receipt and transmission of messages;
- T_4 is in charge of copying the value received in the input of the Business Logic and preparing the output for the transmission.
- T_5 is the application thread that contains the logic of the system.
- T_6 is a diagnostic thread;
- T_7 is a USB driver used for logging data in a key.

The scheduler operates by dividing processor time into epochs. Within each epoch, every task can execute up to its time slice. In this case, the scheduler has two epochs of 100 ms and the taskset have the following constraints:

- The total time of scheduling cycle is 200 ms.
- Each epoch needs to last exactly 100 ms.
- The first execution of T_3 in the first and second epoch must terminate within 95 ms.
- The second execution of T_3 in the first and second epoch must terminate within 95 ms.

- The second execution of T_3 in the first and second epoch must execute at least 65 ms after the first one.
- T_4 in the first epoch must terminate within 90 ms and in the second epoch in 140 ms.
- The total processor time assigned to T_5 in the two epochs must be of at least 90 ms.

The taskset used in our example is defined in the Tables 1 and 2 with the relative scheduling order and parameters.

Table 1. TaskSet in first epoch.

Epoch1	R_i	C_i	D_i
T_1	0	6	6
T_2	6	5	11
T_3	11	16	27
T_2	27	5	32
T_3	32	4	36
T_4	36	24	60
T_5	60	40	100

Table 2. TaskSet in second epoch.

Epoch2	R_i	C_i	D_i
T_1	0	6	6
T_2	6	5	11
T_3	11	16	27
T_2	27	5	32
T_3	32	4	36
T_5	36	40	76
T_4	76	8	84
T_6	84	10	94
T_7	94	6	100

The constraints and parameters given for the taskset are the basis on which a model of the scheduling algorithm can be built. We resorted to the use of Petri Nets, that result quite intuitive in the modelling of scheduling algorithms [5,10,15,23]. In order to represent time, we have investigated the use of both-Timed Petri Nets (TPN) [20] and Coloured Petri Nets (CPN) [13]. In the following we illustrate the two kinds of models by means of this running example, giving a comparison between the two modelling approaches.

4.2 Presentation of Fixed Scheduler Models

In Fig. 2 the model generated with the tool TINA [4,22] for a fixed scheduler [1,3,11] is reported. As we can see the representation with TPN is a little bit chaotic and representing larger sets of tasks could be very difficult. Looking at the model we can underline some diagram parts which are used for the verification of constraints [23]:

- **Check for the Total Time**
  The network used to control the time of each epoch consists of two transitions and respectively five and three places. Taking into consideration the network (a) in Fig. 3, the transition t27 counts the total time available for the execution in the epoch. When the available time expires, the token content in place p34 is moved to place p35 inhibiting the passage of the token coming from the last running thread to places p36, p62 and p30.
  If this happens it means that the execution time has not respected the constraints for this epoch. If, instead, the execution ends before the deadline, the transition t28 will not be inhibited by place p35 and will allow tokens to go in places p36, p62 and p30, establishing the positive conclusion of the first epoch and the start of the second one.
- **Check Constraints on T_3**
  The network in Fig. 4 models the various checks on the execution times for T_3. For example the last block checks that between the first execution of T_3 in the first epoch and the second execution in the second epoch, at least 65 ms have expired. The transition t25 is enabled when the task is running and, if the task completes before the time set in the transition t23, scheduling can continue. Otherwise, if the task does not complete within the specified time, the inhibitor arc starting from p65 does not allow the scheduler to continue.
- **Checking the Scheduled Time between Two Epochs**
  The network in Fig. 5 monitors the execution time of a task between the two epochs. The transitions t14 and t17 are enabled when the task is run in both the first and the second period. This starts the timer of transition t2. If the task completes before the time set in the transition, the scheduling can continue. Otherwise, if the task does not complete within the specified time, the inhibitor arc starting from p19 does not allow the scheduler to continue.

The simulation of this model by means of the TINA tool ends either with a token at place p54, which means that the hypothesized schedule is correct, or by stopping as soon as an error is generated, with a different marking.

We show now the corresponding model described as a CPN. In Fig. 6 the model of the running example generated with CPN tools 4.0 [8,14] is reported. As we can see the representation with CPN is more compact than the one seen with TPN, for example by using only one place we can represent all the tasks of the set. The tasks are represented as a list of objects, and each one is represented by a token having as colour two attributes: a string that contains the name and one integer that represents the WCET C_i of the task.

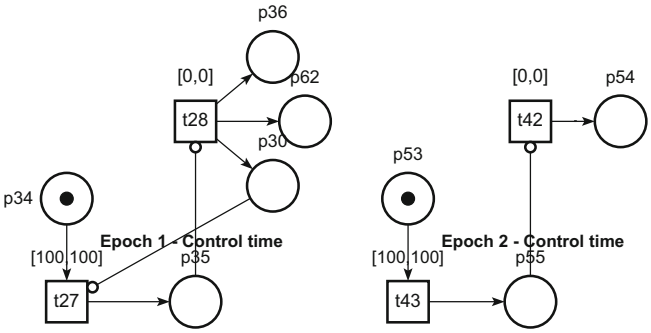

Fig. 3. Diagram of epoch control block.

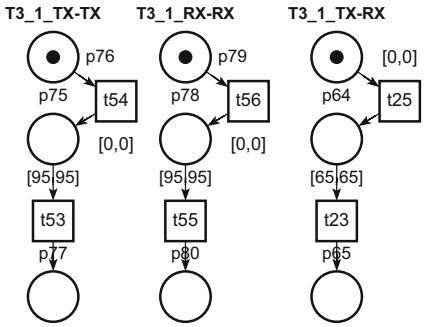

Fig. 4. Diagram of block for the verification of constraints on task T_3.

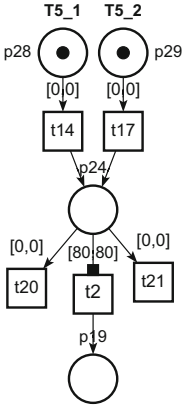

Fig. 5. Check block for the scheduled time between two epochs for task T_5.

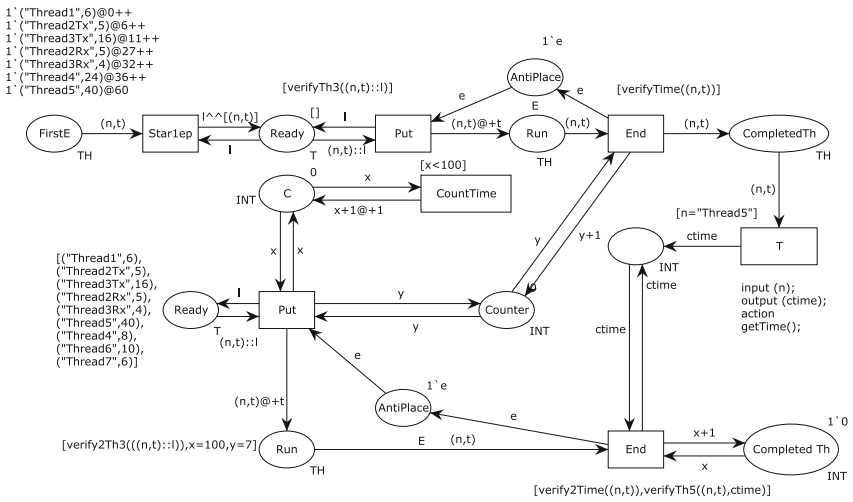

Fig. 6. Coloured Petri Net model for a fixed scheduler.

Inhibitor arcs are not provided by CPNs (as supported by CPNTools); since they are extensively used in our modelling, we have used a pattern that allows to simulate their behaviour: the *Antiplace* pattern, natively provided by CPNTools. This is exemplified in Fig. 7, where the execution of a thread at a certain time is achieved by simulating an inhibitor arc on place Run by using the Antiplace pattern () initialized with a token: when a thread is executed, the token is removed from Antiplace, so that the transition Put is not enabled until the thread finishes executing (so the token spends in place Run a time equal to its value C_i, represented by the variable t), and then it enables the transition End; at this point a token is put back in Antiplace, allowing the next thread to run. In Fig. 6 the Antiplace pattern is used in the modelling of both the first epoch (top Antiplace) and of the second (bottom Antiplace).

The second epoch performs its scheduling after 100 time units have elapsed. Time is not inherently modelled in CPNs as is in Timed Petri Nets. Hence a time-passing simulating pattern has been used; the pattern shown in Fig. 8 implements a timer that increments by 1 at each simulation step, till the simulated time has reached 100, in which case the transition CountTime is disabled. The Timer pattern has been used in Fig. 6 to start the second epoch at time 100. The place C containing a token value 100 allows threads to run, as long as the other constraints on the transition Put are respected; in particular the threads of the first portion of the schedule (first epoch) must have all finished running.

Due to the absence of the built-in timing mechanisms of TPNs, verification of the constraints on the execution time of the thread need to be explicitly realized by means of some functions listed on the transitions. On the first and second transitions named "Put", for example, we can find respectively the functions called *[verifyTh3 ()]* and *[verify2Th3 ()]*. These two functions implement the

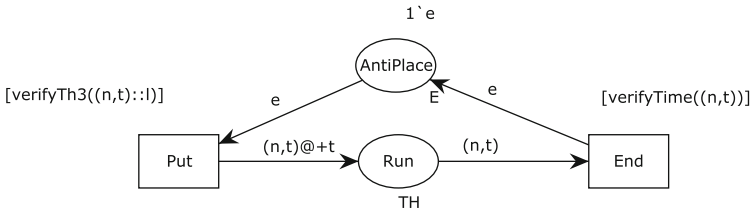

Fig. 7. Antiplace pattern used to simulate an inhibitor arc.

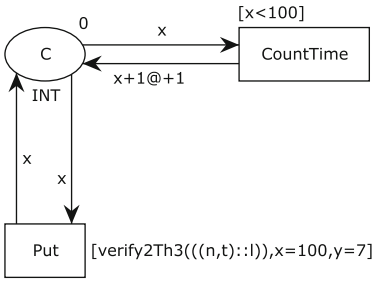

Fig. 8. Timer pattern.

constraint that between the two executions of T_3 cannot elapse less than 65 ms. The functions are defined as follows:

```
fun verifyTh3((n,t)::l) =
  if  n="Thread3" andalso
      intTime() > 65
  then false else true
fun verify2Th3(((n,t)::l)) =
  if  n="Thread3" andalso
      (intTime()-100) > 65
  then false else true
```

The function checks if the token in input to the transition represents the task 3, and verifies that the current simulation time (obtained with the function *intTime()*) is less than 65 units. If the constraint is not respected, the transition is not enabled.

On the transition "End" we can find a function named *[verifyTime()]* that checks all the other constraints (the function is similar to the one above).

An exception is the constraint on T_5 that is represented by function *[verifyTh5ctime()]* placed as guard on the same transition. The modelling of this last constraint, specific for task T_5, requires to save in a variable the time at which the token of the T_5 exits from the "Run" place in the first epoch. This has been achieved through the transition "T" with label pattern *input, output, action* where we take a variable in input (variable n) and by the action (*getTime()* function) we generate an output (variable *ctime*). This transition is

enabled only for T_5 as we can see from the guard on the arch. So the variable that we have obtained has be used in the function:

```
fun verifyTh5 ((n, t), ctime)=
  if  n = "Thread5" andalso
      (intTime () - ctime) >= 90
  {then false else true}
```

Similarly to the modelling done with TPN, also the simulation of the CPN model by means of the CPN Tools 4.0 stops if one of the constraint is not satisfied, so the user is able to understand where the problem is located.

5 Modelling the Runtime Scheduling

As previously said, the platform also implements a runtime (on-line) scheduling algorithm: a round robin scheduling with priority levels, deadlines, a preemption mechanism and a donation mechanism. Runtime scheduling is used for applications of the platform that do not require stringent hard real-time requirements. Although scheduling predictability in these applications is less urgent, we have applied the same modelling framework used for the pre-runtime algorithm to this case: having a certain level of predictability at a low cost can anyway avoid annoying (although not safety-critical) software bugs due to poor scheduling performances, that could anyway increase software maintenance costs.

Also in this case we have used both TPNs and CPNs in order to complete the comparison of the two modelling frameworks also in this other case. The following sections provide the generated models with Petri Nets for three variants of the Round Robin algorithm, namely with FIFO queue, with prioritized FIFO queue, and adding preemption. The experiments are conducted on a reduced taskset of three tasks, starting from the simpler variant, by inserting then various functions incrementally.

5.1 Round Robin with FIFO Queue

The first variant considered is a round robin without priority, without preemption but with the introduction of a FIFO queue for arriving tasks. Figure 9 shows the diagram of execution time for this variant, assuming the following taskset data:

- The first process arrives at time 20 and has a duration of 8 time units,
- The second arrives at time 25 and has a duration of 12 time units,
- The third arrives at time 20 and has a duration of 16 time units.

The time slice assigned to each task at run time is 4 time units. In the Timed Petri net of Fig. 10 a FIFO queue for the management of the processes during their arrival and their displacement has been introduced in the *WAIT* places. Three FIFO queues have actually been implemented (highlighted by a box), one for each task, given the impossibility to use only one for all the tasks, due to the fact that in TPN it is not possible to distinguish tokens representing different tasks.

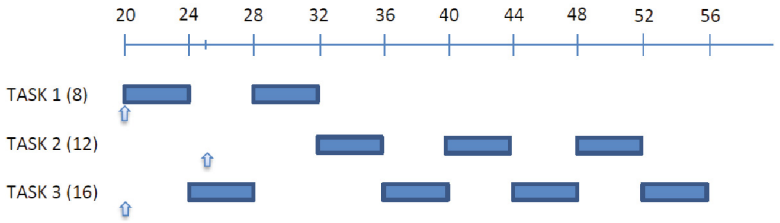

Fig. 9. Temporal schema of a RR with three tasks with FIFO queue.

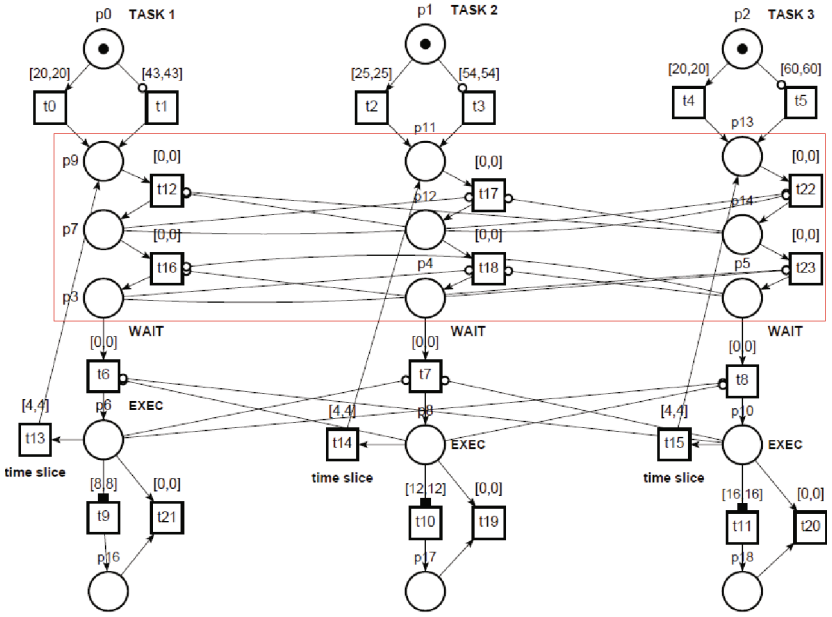

Fig. 10. Timed Petri Net relative to a RR with three tasks with FIFO queue.

Each queue is formed by three places and two transitions. The places are used to store the position of the task in the queue and transitions allow the progress of the task in the queue, moving the token. To simulate the correct order of tasks in the queue, inhibitors arcs have been used, which inhibit the passage of the token to the next place if the other queues already contain a token in a place of the same level.

The same round robin variant was modelled with CPNs and the result is shown in Fig. 11. The tasks are represented by tokens of type *Sting*int* defined as: *colset T = product STRING*INT timed;*, where the string is the task identifier and the integer represents the time slice. The FIFO queue in this case can be programmed as a single token having as type a list structure: the management

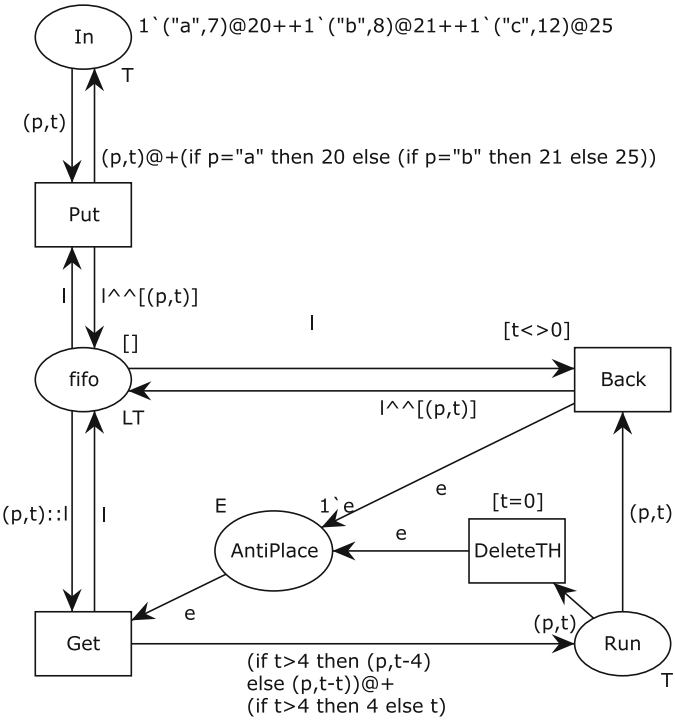

Fig. 11. Coloured Petri Net relative to a RR with FIFO queue, and three tasks.

of the queue is represented by the place *fifo*, which takes token type LT defined as: *colset* $LT = list\ T\ timed$ or rather an object list of type T.

Tokens have an initial timestamp representing their time of arrival. They are put in the list by the concatenation function $l\hat{}\hat{}[(p,t)]$, where l is the token associate to the place *FIFO*. If there are no tokens in the place *Run*, the transition *Get* is enabled, the element at the head of the FIFO is extracted by the function $(p,\ t) :: l$, and the updated list is sent back to the place *FIFO*. Before being added to *Run* every input token receives a timestamp, and the time slice value is decremented. If the remaining time is less than the timeslice, it receives a timestamp value equal to the remaining time, and the integer value of the token, represented by the variable t, is brought to 0. In this way, once elapsed the timestamp, the token will not be placed back in the queue but will be eliminated through the transition *DeleteTh*.

5.2 Round Robin with Priority FIFO Queue

Figure 12 shows the timing diagram of a Round Robin scheduling with FIFO queues to which priority has been added. Similarly to the previous example, there are three processes:

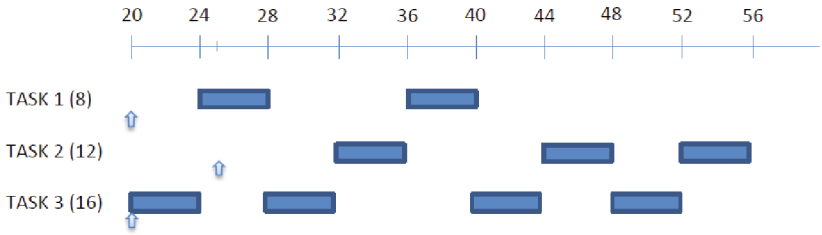

Fig. 12. Temporal schema of a RR with three tasks with FIFO queue and priority.

- The first arrives at time 20 and has a duration of 8 time units,
- The second arrives at time 25 and has a length of 12 time units,
- The third arrives at 20 and has a duration of 16 time units.
- The second and third process have equal priority and greater than the first one.

The quantum of CPU time assigned to each of them at run time is 4 time units.

The Timed Petri Net of Fig. 13 is the model of the round robin variant with priority and FIFO queue. The boxes show queues formed by nine places arranged in three rows and connected together with instant transitions. This is a generalization of the previous modelling of a single queue for the three tasks case, where in principle the three tasks can have three different priority levels: the priority levels are represented by the three columns in each box. The task priority is expressed by including a link to the first place of the queue related to the actual task priority: If the arc is connected with the first (last) vertical row of places, the process will have the lowest (highest) priority. In the case under consideration, the first task has a lower priority than the other two tasks, that have equal priority. Notice that only one of the columns is connected, so the other two are useless, but this design is maintained for easy modification of tasks'priorirty. As for the simple FIFO queue of Fig. 10, inhibitor arcs are used to enforce the correct priority and FIFO policy.

The corresponding CPN model for this case is shown in Fig. 14. The model must then insert the thread into a ready queue based on their order of arrival and an integer value representing the priority of the thread. A token that has a priority value higher than those already present in the queue will be inserted in the head, and then perform first. Hence the management of the priority and FIFO policy is implemented through a data structure instead of the convolutes net layout of the TPN case. The structure of the model is virtually unchanged compared to the case without priority, the differences are basically two, namely the label on the arcs in input to the place that represent the tail and a different type to define the task. In this model, threads are represented by token type *colset T = product STRING*INT*INT timed* where last INT value is associated to the task priority. In the previous model the insertion was performed by concatenation function and the token was placed at the tail of the queue. In the version with priorities a check on the value that represents the priority is needed

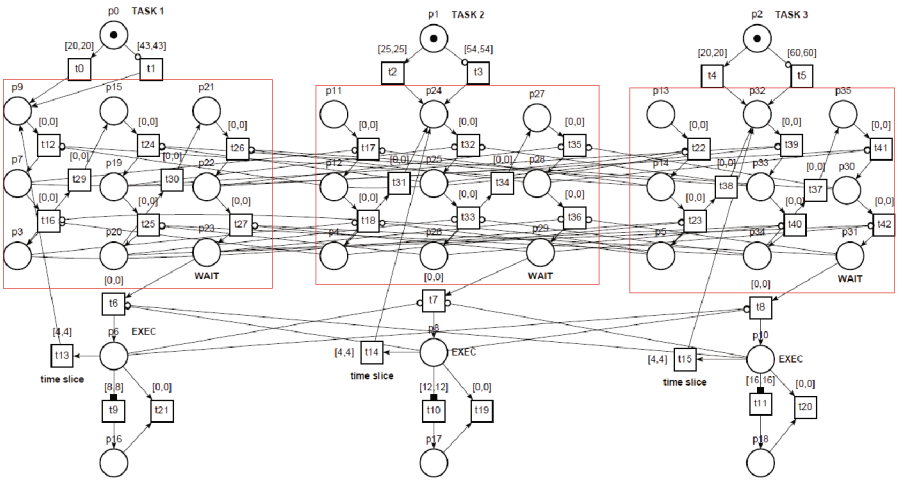

Fig. 13. Timed Petri Net relative to a RR with three tasks with FIFO queue and priority.

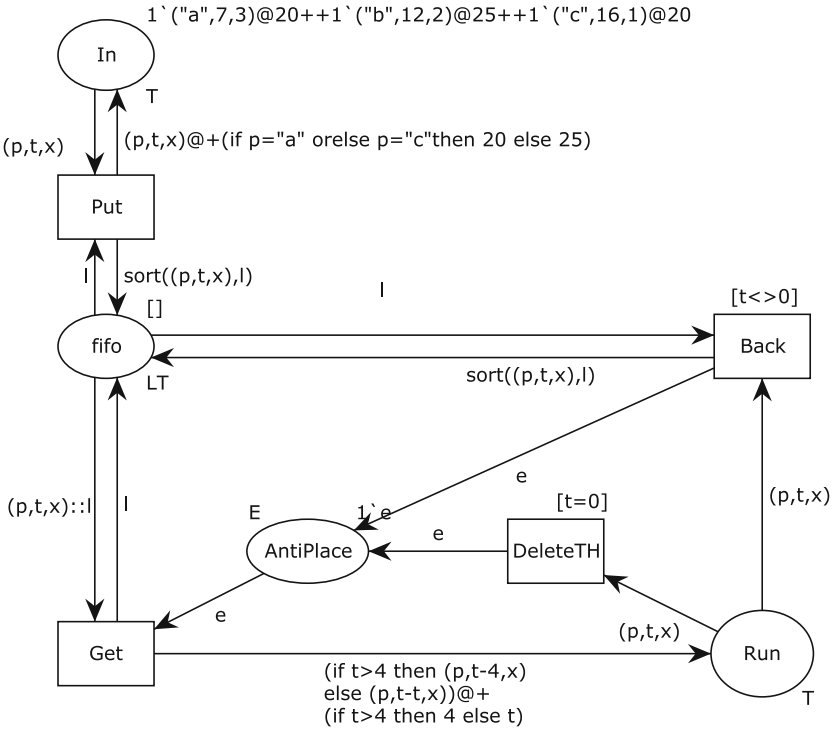

Fig. 14. Coloured Petri Net relative to RR scheduling model with with FIFO queue and priority.

in order to determine the location in the queue, and this is achieved with the sort function *sort((p, t, x), l)* defined as follow: $funsort((p,t,x),[]) = [(p,t,x)]|$
$sort((p,t,x),((m,s,q)::l)) =$
$if higherPr(x,q) then (p,t,x)::(m,s,q)::l \qquad else (m,s,q)::(sort((p,t,x),l));$

where *higherPr(x,q)* is another function that performs a comparison between two integer values that represent the degree of priority and returns a boolean value, thus defined: $funhigherPr(x,y) = (x > y);$. The *sort()* function takes as parameters the variables (p,t,x), respectively, of type $String, int, int$ that represent the three attributes of a thread, and the variable l of type LT, which represents the list. If l matches the empty list, the token will be inserted in the list, otherwise, will run the function *sort ((p, t, x), ((m, s, q) :: l));* that makes the comparison with the element that is currently leading the list. If the priority value of the token to be inserted is higher than that in front of the list, then it will be placed in front of the latter. Instead, if the value is lower, the *sort()* function will be called through that list, with the exception of the head element in the head, recursively until it finds a token with a lower priority or the list is empty.

5.3 Round Robin with Priority FIFO Queue and Preemption

The last variant is a round robin scheduling with priority FIFO queues and preemption, whose time schema is given in Fig. 15. We use the same example taskset data of the previous variant.

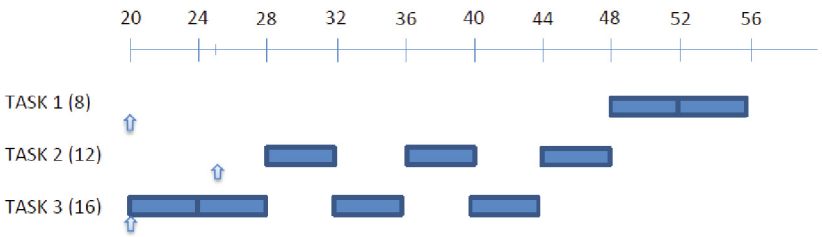

Fig. 15. Temporal schema of a RR with three tasks with FIFO queue, priority and preemption.

The Timed Petri Net of Fig. 16 adds to the previous model the preemption technique: the box highlights the transitions designed for this purpose. If a task of higher priority arrives in *WAIT* place, transitions *t28* and *t43* are activated, triggering the move of the token representing task 1, with lower priority, from the place *EXEC* to *WAIT*, which represents preemption.

The model of this variant by means of CPNs is shown in Fig. 17. Compared to Sect. 5.2 a transition *CheckPr* and a place called *Count* have been added.

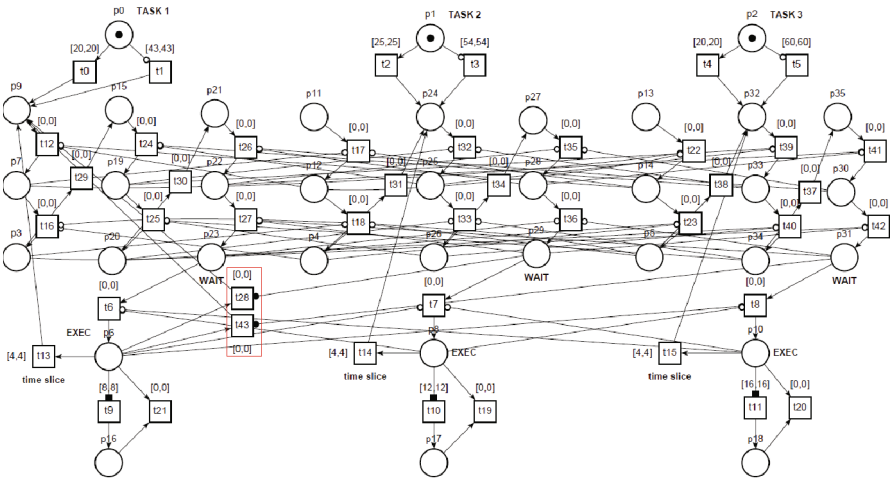

Fig. 16. Timed Petri Net modelling a RR with FIFO queue, priority and preemption, with three tasks.

The place is used to simulate the continuous flowing of time, with time increasing of a time until at each simulation step. Indeed, the model of Sect. 5.2 advances the time at each simulation step of the amount of time needed to reach a change of system status, while preemption requires to check the status of the tasks at any simulation step. When the integer token contained in *Count* reaches four (the value attributed to the timeslice), it enables the transition *Back in Run* and the token is queued by the *sort()* function. For every unit of time, through the transition *CheckPr* a comparison is made between the priority of the running threads and that of the thread at the top of the queue. If the priority of the latter is lower, the token currently in Run will decrease the execution time of 1 and its timestamp will be incremented by 1. The transition *CheckPr* remains enabled as long as the executing thread will have an execution time greater than 0 or until it will have spent the whole time slice. In case a token is in the *Run* place and a token with higher priority arrives in the queue, the time counting is stopped and the replacement is done instantly, by inserting the token with the higher priority in *Run* and inserting the other in the queue via the *sort()* function. The verification of the priority value is executed, via inscriptions on the arcs, by the function *higherPr(x, y)* defined earlier. The same function is used to determine the value of the timestamp on the token in *Run* and *Count*, and to decrease or not the execution time of the thread. If in fact the function returns true, it means that they will be replaced, and the value of the token t in execution will not be decreased.

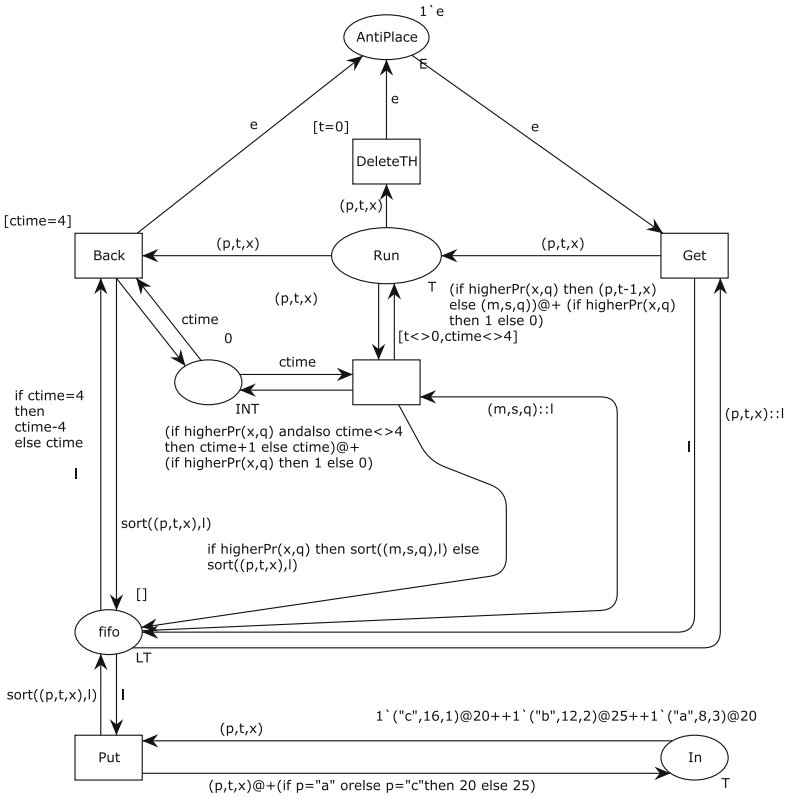

Fig. 17. Coloured Petri Net relative to RR scheduling model with FIFO queue, priority and preemption.

6 Comparison Between TPN and CPN

The experiments have allowed a comparison between the two Petri Nets dialects and related supporting tools, in particular enlightening the following points:

The TPN model is difficult to read, and the addition of further tasks would result in a huge increase of the places and transitions number, making it more and more unreadable. This increase is due to the following reasons:

- Any place can hold a single token and the execution of a thread must be reproduced a number of times equal to the number of modelled processes; indeed in TPNs it is not possible to express an attribute that differentiates the identity of a token.
- Time management for each thread is left to time constraints on the transitions themselves.
- It is not possible to create aggregate objects: a FIFO queue, for example, can be realized only through checks by inhibitors arcs with a number of places

that depends on how many threads should be modelled (the number of places to represent the queue is equal to n^2 where n is the number of threads).
– With the inclusion of the priority, it is noted that for each thread nine places and seven transitions with inhibitor arcs between them are needed for three thread. In this case we have a cubic relation of the net size to the number of threads, although optimizations can be done by loosing flexibility of the approach.

The CPNs instead can represent a queue using a single place that contains a token of type list. Time management is shifted to the token colour using an integer and a timestamp. This allows a large number of tasks to be represented by simply adding tokens to the initial marking, leaving the structure of the model unaffected.

As cons the CPN Tools software does not support the inhibitors arcs, so it was necessary to simulate them through the Antiplace pattern. The increase of the size due to the use of this pattern is however only locally additive, and the absence of replicated instances of inhibitor arcs typical of TPN models allows for containing the usage of the pattern to a few units.

CPNs do not natively support time, so time constraints (modelled in TPN through places and transitions) have to be expressed in auxiliary functions, but this in the end simplifies the model.

With TINA and TPNs the management of time during simulation allows to easily understand the global state of the system. The time is increased at any simulation step of a time unit. In CPNTools and CPNs if there are no transitions enabled at the current time, the simulated time count is increased in one step, up to the time at which at least one transition is activated. In one case we had to enforce simulation of step by step time advance by means of a specific Timer mechanism.

CPNs however resulted to be more advantageous in terms of time spent in model design or in changes, mainly for two reasons:

1. constraints can be simply modelled by a guard on the transition, expressed by a function written in pseudo code, which is easier to express;
2. populating the model with new tasks does not require to draw new graphic elements but just add an entry to the related place;

We have experienced that the time spent in CPN modelling is at the end less than half that spent in TPN modelling.

7 Conclusions

We have applied the two modelling options sketched above to different scheduling algorithms, a fixed one and a Round Robin, and different sets of tasks as well. The quite straightforward conclusion is that the CPN modelling is more advantageous in terms of size and readability of the model, and in terms of adaptability of the model to different task sets.

It is indeed easier with CPN to instantiate the same model, for the same scheduling algorithms, on a different set of tasks, and this is what is important in the daily application of this modelling framework. Since essentially only the taskset data need to be changed for a new, or modified, specific application, the overall time to analyse a new taskset, summing up the time to produce a model of the schedule of a new specific application, to run a simulation and to analyse the simulation, is about two hours with a TPN modelling and about one hour with the CPN modelling. Anyway, this time compares with the much longer time (eight hours) needed by the previously used empirical approach, and therefore is convenient in both cases.

Even if some rework is needed in case of a negative response of the simulation, the information returned by the simulation helps understanding where the problem lies, indicating the solution to the problem. Usually one rework cycle is at most needed, so the overall cost is anyway reduced.

For this reason we have not considered convenient to investigate solutions based on counterexample generated by a model checker [12], able to provide automatically the taskset parameters satisfying the scheduling requirements.

The low cost of the simulation based solution has an obvious positive impact on the costs of the process of instantiating a generic application to a new specific application for marketing a new product or variant.

Regarding the Round Robin runtime scheduling algorithm, we have shown the modelling, with the two Petri Net variants, for a taskset of three tasks. It is already evident from the presented models that for real application tasksets, such as one that we have addressed, containing 16 tasks, with 32 priority levels, the TPN model cannot be feasible, while the CPN model is an easy extension of the one presented.

The design process based on this modelling approach is currently under experimentation by our industrial partner, with the aim of introducing it in the routine customization process. An help for this introduction could come from providing tools to support an easier instantiation of the generic models into specific ones, so that the use of CPN is transparent to the final user who only sees the simulation results. This objective requires also a facility to explain the reasons of a negative response without showing the underlying CPN model. This is considered as future work.

Although motivated by specific needs of a railway signalling company, we believe that this approach can be ported to other domain as well, as soon as configurable real-time applications have to be designed on top of available real-time scheduling algorithms.

Acknowledgements. We wish to thank Marco Bartolozzi, Daniele Marchetti and Luca Santi for their contribution to the conducted modelling experiments.

References

1. van der Aalst, W.M.P.: Petri net based scheduling. Oper. Res. Spektrum **18**, 219–229 (1996). Springer

2. Alur, R., Dill, D.: The theory of timed automata. In: Bakker, J.W., Huizing, C., Roever, W.P., Rozenberg, G. (eds.) REX 1991. LNCS, vol. 600, pp. 45–73. Springer, Heidelberg (1992). doi:10.1007/BFb0031987
3. Barreto, R., Cavalcante, S., Maciel, P.: A time Petri Net approach for finding preruntime schedules in embedded hard real-time systems. In: Proceedings of Distributed Computing Systems Workshops, pp. 846–851. IEEE (2004)
4. Berthomieu, B., Vernadat, F.: Time petri nets analysis with TINA. In: Quantitative Evaluation of Systems, pp. 123–124. IEEE (2006)
5. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent system using time petri nets. IEEE Trans. Softw. Eng. **17**(3), 259–273 (1991). IEEE
6. Buttazzo, G.: Hard Real-Time Computing System, 3rd edn. Springer, New York (2011)
7. Cenelec: Cenelec EN 50128:2011. In: Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems (2011)
8. CPNTools (2015). http://cpntools.org/
9. Fantechi, A.: Twenty-five years of formal methods and railways: what next? In: Counsell, S., Núñez, M. (eds.) SEFM 2013. LNCS, vol. 8368, pp. 167–183. Springer, Cham (2014). doi:10.1007/978-3-319-05032-4_13
10. Felder, M., Mandrioli, D., Morzenti, A.: Proving properties of real-time systems through logical specifications and petri net models. IEEE Trans. Softw. Eng. **20**(2), 127–141 (1994)
11. Grolleau, E., Choquet-Geniet, A.: Off-line computation of real-time schedules using Petri Nets. Discrete Event Dyn. Syst. **12**(3), 311–333 (2002). Springer
12. Gardey, G., Lime, D., Magnin, M., Roux, O.H.: Romeo: a tool for analyzing time petri nets. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 418–423. Springer, Heidelberg (2005). doi:10.1007/11513988_41
13. Jensen, K.: Coloured petri nets. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) ACPN 1986. LNCS, vol. 254. Springer, Heidelberg (1987)
14. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3), 213–254 (2007). Springer
15. Leveson, N.G., Stolzy, J.L.: Safety analysis using Petri Nets. IEEE Trans. Softw. Eng. **13**(3), 386–397 (1987)
16. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973). ACM
17. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). IEEE
18. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall PTR, Upper Saddle River (1981)
19. Petri, C.A.: Kommunikation mit automaten. Ph.D. thesis. Universitat Hamburg (1962)
20. Ramchandani, C.: Analysis of asynchronous concurrent systems by Timed Petri Nets. Massachusetts Institute of Technology (1974)
21. Stankovic, J.: Misconceptions about real-time computing. IEEE Comput. **21**, 10–19 (1988). IEEE
22. TINA (2015). http://projects.laas.fr/tina/
23. Tsai, J., Yang, S.J., Chang, Y.-H.: Timing constraint Petri Nets and their application to schedulability analysis of real-time system specifications. IEEE Trans. Softw. Eng. **21**(1), 32–49 (1995). IEEE