



Modelling and Schedulability Analysis of Real-time Sequence Patterns using Time Petri Nets and Uppaal

Angelo Furfaro and Libero Nigro

Laboratorio di Ingegneria del Software,
Dipartimento di Elettronica Informatica e Sistemistica,
Università della Calabria, 87036 Rende(CS) - Italy
Email: a.furfaro@deis.unical.it, l.nigro@unical.it
Web: <http://www.lis.deis.unical.it>

Abstract. This paper proposes an original approach to the schedulability analysis of real-time systems specified by Time Petri Nets (TPNs). The focus is on sequence patterns of transition firings (execution tasks). A TPN model is first translated in the Timed Automata terms of the popular UPPAAL tool. Then schedulability properties of tasks are verified through reachability analysis. The approach is efficient and scalable. The paper demonstrates the concrete application of the approach through examples. Finally, conclusions are drawn together with an indication of on-going and future work.

Keywords: real-time systems, time Petri nets, timed automata, Uppaal, schedulability, firing sequence patterns

1 Introduction

Time Petri Nets (TPNs) [16] have been proven to be a very convenient tool for specifying timing constraints in time-dependent systems [6, 22, 10]. TPNs associate time pairs with transitions instead of a single delay as in timed Petri nets [18], thus TPNs are more general than timed Petri nets [4]. TPNs support formal analysis through an adaptation [4, 6] of the well-known reachability analysis technique of Petri nets [17]. A reachability graph represents the complete dynamic behaviour of a TPN based on the interleaving semantics. Each node of the graph is a *state class*. Edges are labelled by transition firings. A state class holds a *marking* and a *firing domain* reflecting timing constraints about *when* the state class is possibly reached in the time domain. Examples of concrete tools permitting the enumeration of the state classes of TPNs and enabling reachability analysis are TINA [5, 20] and ROMEO [12, 19].

As pointed out in [22], although schedulability is closely related to reachability, it has more specific concerns with transition sequences rather than markings or states. In particular, an end-to-end delay of a task execution, that is an important issue in time critical systems, cannot directly be derived from firing domains and state classes.

In [22] a method is proposed for making schedulability analysis for a subclass of TPNs referred to as “well-structured” (WS) TPNs. In a WS TPN any firing sequence leading from the initial marking M_0 to a given reachable marking M_n , can be built by composing firing sequences taken from a finite set of elementary sequences including basic loops (see later in this paper for an example).

This paper proposes an original approach to schedulability analysis of sequence patterns of real-time systems abstracted by TPNs. First a TPN model is translated into Timed Automata (TA) [2] in the context of the UPPAAL tool [3, 21]. Then TCTL model checking on the achieved TA product system is used, possibly with the help of *decoration* means [15]. Translation is assisted by a customization of the TPN DESIGNER toolbox [7] which supports various kind of time-extended Petri nets, e.g. GSPN-like. TPN DESIGNER allows one to (i) graphically draw a TPN model (ii) simulate the model for preliminary behavioural/timing checking (iii) generate an equivalent TA representation of the model to be used for exhaustive verification in UPPAAL. The translation rests on a single TA template associated with generic TPN transition. The translation is similar to that described in [8] but was independently achieved. Motivations for choosing UPPAAL instead of existing special-case TPN tools are related to the desire of exploiting a popular tool which ensures an efficient and scalable approach.

The paper is structured as follows. Section 2 introduces TPN modelling concepts and describes a real-time model example together with its firing sequences (execution tasks) schedulability aspects. Section 3 summarises the translation process from TPNs to TA/UPPAAL and applies it to the chosen model example. A *decorator* automaton is proposed which enables task schedulability analysis. More general handling of sequence patterns in which the first transition can occur multiple times is then discussed. Section 4 gives some information about the effectiveness of the proposed approach and translation tool. Finally, conclusions are presented along with an out look to on-going and future work.

2 TPN Modelling and Schedulability Concerns

2.1 TPN Concepts

In this work Time Petri Nets are assumed augmented by inhibitor arcs. A TPN is a tuple $TPN = (P, T, B, F, I_{nh}, M_0, I)$ where P and T are non empty finite disjoint sets respectively of places and transitions of the underlying Petri net [17]; B is the backward incidence function: $B : T \times P \rightarrow \mathbb{Z}$, with \mathbb{Z} the set of integers; F is the forward incidence function: $F : T \times P \rightarrow \mathbb{Z}$; I_{nh} is a set of inhibitor arcs: $I_{nh} \subseteq P \times T$; M_0 is the initial marking: $M_0 : P \rightarrow \mathbb{N}$, i.e. the non negative number of tokens assigned to places; I is the static firing interval function, $I : T \rightarrow \mathbb{N} \times (\mathbb{N} \cup \{\infty\})$. $B(t, p) = w > 0$ if there is an arc with weight w from input place p to transition t . $B(t, p) = 0$ for a non existing input arc (t, p) or for an inhibitor arc $(p, t) \in I_{nh}$. An inhibitor arc is graphically represented by a dot terminated line. $F(t, p) = w > 0$ if there exists an arc from transition t to output place p with positive weight w . $F(t, p) = 0$ for a non existing output arc

(t, p) . The set of input and inhibitor places of t is said its *preset* (denoted $\bullet t$). The set of output places constitutes the transition *postset* (denoted $t\bullet$). I associates with each transition t a *dense* firing interval whose bounds are supposed to be specified by non negative integers: $I(t) = [a, b]$ with $0 \leq a \leq b$, b can be ∞ . Bound a is said the (static) *earliest firing time* (EFT) of t , b the (static) *latest firing time* of t (LFT).

A transition t is said to be enabled in marking M , denoted by $M[t >$, iff: $\forall p \in P \ M(p) \geq B(t, p)$ if $(p, t) \notin I_{nh}$, $M(p) = 0$ if $(p, t) \in I_{nh}$. Let τ be an instant in time when transition t is enabled. Provided t is continuously enabled, t cannot fire before $\tau + a$ but *must* fire (*strong firing model*) before or at $\tau + b$, unless it is disabled by the firing of another (conflicting) transition. Transition firing semantics is best understood by using the *server* analogy and by thinking about of an hidden *timer* (*clock*) associated with the transition. As soon as the transition gets enabled, the timer is reset and the underlying server starts working (transition under firing). Let θ be the elapsed time since the enabling. A *time-strict* transition ($b < \infty$) can complete its firing when $a \leq \theta \leq b$, whereas for a non time-strict transition $a \leq \theta < b$. Whether a transition can complete its own firing depends in general on the firing condition of all the other transitions in the model.

When the transition completes its firing, tokens are removed from $\bullet t$ and new tokens are generated into $t\bullet$ as in classical Petri nets. Let M_{before} be the net marking when t fires. The firing of t transforms M_{before} in M_{after} , denoted by $M_{before}[t > M_{after}$, by an instantaneous and atomic process in two phases:

1. *token withdrawal*
 $\forall p \in P, M'(p) = M_{before}(p) - B(t, p)$
2. *token deposit*
 $\forall p \in P, M_{after}(p) = M'(p) + F(t, p)$

M' represents the intermediate marking generated after token withdrawal. A transition t_p is said *persistent* to the firing completion of transition t if t_p is enabled in M_{before} , and maintains its enabling condition also in M' and in M_{after} . A non persistent transition can lose its enabling either at M' or at M_{after} . Following the token deposit phase one or more transitions can become enabled. They are said *newly enabled* transitions. After its own firing, if t is still enabled, it is considered a newly enabled transition (*single server semantics*).

2.2 A Modelling Example

In the following an assembly subsystem of a flexible manufacturing system (FMS), adapted from [22], is modelled and its schedulability concerns highlighted. An FMS is a real-time system composed of a number of computer controlled tools and automated material handling, assembly and storage systems that operate in an integrated way under the control of host computers. FMS design is challenging in terms of control and scheduling, due to growing demand for high performance and flexibility despite the presence of interlocking factors of concurrency, deadline-driven activities and real-time decision making.

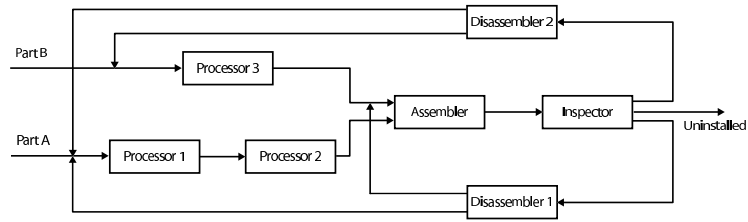


Fig. 1. An FMS assembly subsystem

In the chosen example (see Fig. 1) there are three processors, one assembler, one inspector and two disassemblers. The system receives two type of raw parts (A and B) in input. After processing the input parts, A-part and B-part are assembled in a final product. A-parts are processed in sequence by processor 1 and processor 2. B-parts are processed by processor 3. The inspector is responsible for quality control assessment of assembled products. If an assembled product satisfies quality requirements, it is delivered as a finished product. Otherwise the product is disassembled either by disassembler 1 or 2, depending upon their status. Disassembler 1 extracts A-parts which are sent back to processors 1 and 2, while B-parts are instead sent directly to the assembler. Disassembler 2 generates A-parts which are sent back to processors 1 and 2 and B-parts which are input to processor 3. All components in Fig. 1 have certain timing constraints imposed upon them. According to timing constraints, it is necessary to check different assembling schedules. Whether an A-part and B-part are assembled within a given time, mostly depend on the analysis of the following three cases:

1. neither A-part nor B-part has any quality problem
2. there is no problem with B-part, but A-part cannot pass quality control for m times ($m > 0$)
3. A-part and B-part cannot pass quality control respectively for m and n times ($m > 0, n > 0$).

Fig. 2 shows a TPN model of the assembly subsystem along with the assumed timing constraints. Transition t_0 models the reception of both A-part and B-part. Transitions t_1 , t_2 and t_3 model respectively processor 1, 2 and 3. Transition t_4 represents the assembler, whereas t_5 denotes the inspector. Transition t_6 expresses unloading of a finished product, which can go in input to another subsystem etc. Transition t_7 models the disassembler 1, whereas transition t_8 models the disassembler 2. The initial marking of the model in Fig. 2 mirrors A-part and B-part are already available for processing. Analysis of case 1 above, means checking the following transition sequences: $t_0\sigma t_4 t_5 t_6$ where σ , due to the concurrent paths $t_1 t_2$ and t_3 , can be: $\sigma = t_1 t_2 t_3$ or $t_1 t_3 t_2$ or $t_3 t_1 t_2$. Checking case 2 requires analysing the following transition sequences:

$$t_0 \sigma t_4 t_5 \sigma_1^m t_6, \text{ where } \sigma_1 = t_7 t_1 t_2 t_4 t_5.$$

Finally, checking case 3 means analysing sequences (roughly) expressed by:

$$t_0 \sigma t_4 t_5 \sigma_1^m \sigma_2^n t_6, \text{ where } \sigma_2 = t_8 \sigma t_4 t_5.$$

It is worth noting that repetitions of subsequences (loops) σ_1 or σ_2 can be interleaved. Places p_{10} and p_{11} serve the purposes of controlling respectively the occurrences of loops σ_1 and σ_2 . Setting the marking of p_{10} and p_{11} to 0, forces straight sequences $t_0\sigma t_4t_5t_6$ to occur. In the general case, p_{10} will have m tokens as its initial marking, whereas p_{11} n tokens.

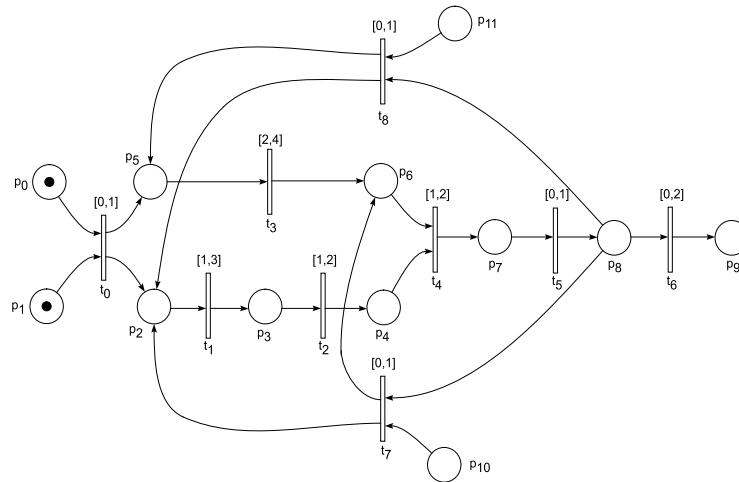


Fig. 2. TPN model of system in Fig. 1

Model in Fig. 2 is an example of a well-structured TPN. Any transition sequence leading to final marking $M_n = \{p_9\}$ can be composed by concatenating basic subsequences as mentioned above. Schedulability analysis of task executions involves both *behavioural* and *timing* verification. Behavioural analysis concerns assessing the functional requirement that each task is effectively schedulable, i.e. it *can* occur. Timing analysis means determining the time span of the entire task and of each belonging transition firing event.

3 Uppaal Translation and Schedulability Analysis

3.1 Mapping TPNs on to Uppaal

UPPAAL [3, 21] makes it possible to verify systems modelled as networks of timed automata (TA) [2] extended with integer variables, structured data types (e.g. arrays) and channel synchronization (CSP-like *rendezvous*).

In the approach advocated in this work the use of TPN DESIGNER [7, 9] is combined with that of UPPAAL. In particular, first a TPN model is graphically drawn in TPN DESIGNER, then it is pre-checked in simulation; finally, the model is translated in UPPAAL as a product of as many TA as there are transitions in the TPN model. Fig. 3 shows the unique proposed template automaton which

reproduces semantics of a TPN transition. The automaton purposely exploits features of latest UPPAAL 4.0.6 version, in particular C data structures, function declarations and loop constructs.

Transition automaton has three parameters: transition unique ID, clock x and a broadcast fire channel for signalling completion of transition firing. Let T and P denote respectively the set of transitions and the set of places in a model, PRE and POST the (maximal) set of input places and output places of any transition. The translation generates a compact representation of the backward $B_{|T| \times |PRE|}$ and forward $F_{|T| \times |POST|}$ incidence matrices (realized as constant data structures) of a TPN model, its marking vector $M_{|P|}$ and the time interval vector $I_{|T| \times 2}$ (constant data structure). An `Info` struct is defined which contains an index in M to select a place, and the weight of an arc linking the place to a transition or vice versa. An index value of -1 is used for a non existing arc. Each element of B or F is an `Info` value. A non strict time interval like $[a, \infty]$ is represented in I as $[a, -1]$. Global functions `bool enabled(const int ID)`, `void withdraw(const int ID)` and `void deposit(const int ID)` respectively check transition enabling, withdraw tokens from input places and generate tokens to output places; they refer to a particular transition by its ID received as a parameter.

Model bootstrapping is achieved by a first synchronization through the broadcast `end_fire` channel, which is launched by the `Starter` automaton shown in Fig. 4. The synchronization allows transitions enabled in the initial marking of the TPN model to move to `Firing/U_Firing` location. Not enabled transitions remain in their `Disabled` initial location. After model bootstrapping, `Starter` will take no part in the subsequent evolution of the model.

An enabled transition starts firing by resetting its clock x and moving to `Firing` if its time interval is strict, or to `U_Firing` if its latest firing time is ∞ . An enabled transition can complete its firing as soon as its clock goes beyond its earliest firing time. A time strict transition is obliged to complete its firing at its latest firing time, provided it is still enabled, by the invariant $x \leq I[ID][1]$ attached to `Firing`. Transition firing, though, is non deterministic among transitions which can complete their firing at current time. Firing completion follows exactly the instantaneous and atomic process in two steps described in section 2.1, which involves committed locations `Withdraw` and `Deposit` in which time is not allowed to pass. The transition first removes tokens from its input places and signals firing completion through the `fire` broadcast channel. Then it moves to the `Withdraw` location where it sends (*first step*) a broadcast synchronization signal over the `end_fire` broadcast channel. This signal forces all other transitions, disabled or under firing, to re-evaluate their enabling status. This is crucial for proper management of conflicts. A no longer enabled transition moves from `Firing/U_Firing` to `Disabled` and resets its clock. The firing process continues by generating tokens in the output places and then by sending (see `Deposit` outgoing edges in Fig. 3) a second broadcast synchronization through `end_fire` (*second step*) which permits detection of newly enabled transitions and non persistent transitions which lose their enabling due to inhibitor arcs. The

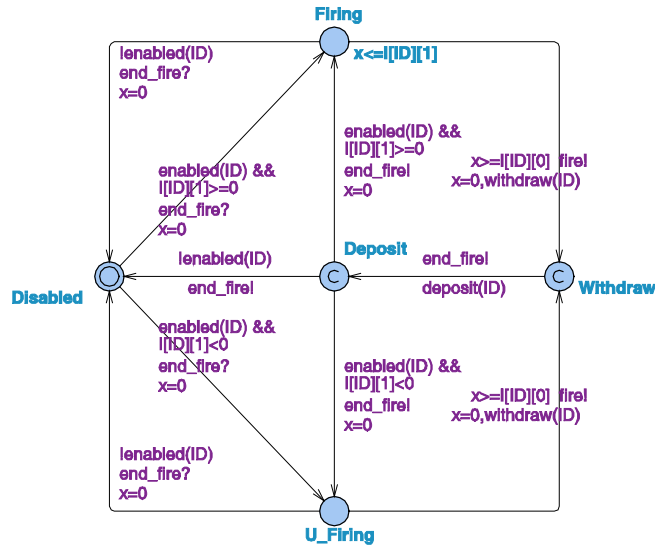


Fig. 3. Transition automaton



Fig. 4. Starter automaton

Transition automaton follows the *single server* firing policy: after its own firing, a still enabled transition is regarded as a newly enabled one and its clock reset.

A TPN model is transformed into the parallel composition of $|T|$ instances of the automaton in Fig. 3 corresponding to the $|T|$ transitions of the source model, and one instance of the Starter automaton. Some *decoration* [15] processes can possibly be added to help schedulability analysis (see later in this paper). Translation correctness can be formally proved by showing that the semantics of the translated UPPAAL model is *timed bisimilar* to that of the original TPN model, by preliminarily mapping both representations on timed transition systems. The formal proof is similar to the one described in [8]. The equivalence of a TPN model and its corresponding UPPAAL model, enables TCTL model checking activities [1] which can directly be interpreted at the TPN level. As for the approach of [8], actually only bounded TPN models can be efficiently verified (checking both safety and bounded liveness properties) using the proposed translation process.

3.2 Schedulability analysis of the FMS model

Being well-structured, the TPN model of Fig. 2 can be analyzed in UPPAAL with the assistance of the Decorator automaton shown in Fig. 5. The Decorator has

paths corresponding to the various transition sequences admitted by the model. The Decorator has distinct locations for each transition firing of a task execution. Loops in Fig. 5 directly correspond to sequence loops in Fig. 2. The Decorator relies on the global fire array of $|T|$ broadcast channels. At system configuration time, each element of the fire vector is associated with a distinct transition.

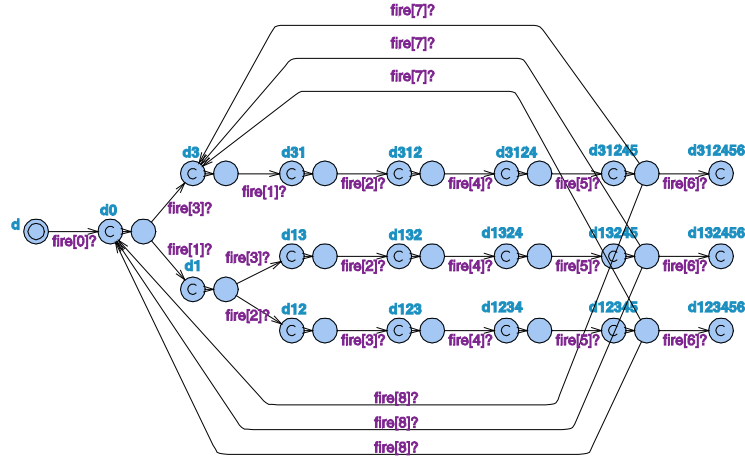


Fig. 5. Decorator automaton for the TPN model in Fig. 2

Verification experiments were separately carried out by checking the three cases in section 2.2. For case 1, the initial marking of places p_{10} and p_{11} (actually the values of $M[10]$ and $M[11]$, M being the marking vector) was set to 0. Behavioural analysis was accomplished by issuing to the UPPAAL verifier “existential” queries like the following:

```
E<> Decorator.d312456
```

which checks that effectively a state of the state graph exists in which the location d_{312456} of the Decorator is reached. Similar queries can be used for locations d_{132456} and d_{123456} which mark other tasks termination. All these queries were found satisfied, which in turn ensured each involved transition is effectively schedulable, i.e. it can fire according to its assigned time interval during task execution. The query

```
Decorator.d --> Decorator.d312456 || Decorator.d132456
|| Decorator.d123456 (1)
```

checks that one of three tasks is always terminated, i.e. that from the initial location $Decorator.d$ it always follows that the Decorator ends in one of its final locations.

After behavioural tests, attention was focussed on the timing aspects of tasks. To this purposes, a global clock z (another decoration feature) was introduced and separately checked on the committed locations of the various transition sequences. For example, the time span of the sequence $t_0t_3t_1t_2t_4t_5t_6$ was determined by using the two *invariantly* queries (to be verified in *every* state of the state graph):

$$A[] \text{Decorator.d312456 imply } z \geq \text{lower_bound} \tag{2}$$

$$A[] \text{Decorator.d312456 imply } z \leq \text{upper_bound} \tag{3}$$

where `lower_bound` is actually the maximum integer which makes true query (1), and the `upper_bound` is the minimum integer which satisfies query (2). The time span of the chosen task which was found to be $[4, 11]$. In a similar way, it was possible to establish the time span of each transition firing (see Table 1). Values in Table 1 fix a few wrong calculations reported in [22]. It should be noted that timing information are relative to the time horizon ($z=0$) when the initial marking of p_0 and p_1 is established.

Similar queries were used for checking cases 2. and 3. of section 2.2. For demonstration purposes, the time span of the model tasks was determined when $m = 3$ (initial marking of $M[10] = 3$) and $n = 2$ (initial marking of $M[11] = 2$). For brevity, only the time span of the three tasks, i.e. the possible firing times of locations `d312456`, `d132456` and `d123456` of Fig. 5, are reported in Table 2. They serve the purposes of answering questions about deadline fulfilment of the manufacturing/production system under loop repetitions. For example, the firing time interval of task $t_0t_1t_2t_3t_4t_5t_6$ was checked by the two queries:

$$A[] \text{Decorator.d123456 imply } z \leq 55 \tag{4}$$

$$A[] \text{Decorator.d123456 \&\& } M[10]==0 \ \&\& \ M[11]==0 \text{ imply } z \geq 18 \tag{5}$$

which were found true only for the shown values. Query (4) which determines the lower bound of task termination, ensures that loop repetitions effectively exhausted.

Obviously, verification can also answer querying the time span of individual transition firings, e.g. when the marking $M[10]$ and $M[11]$ become 0, or in intermediate scenarios.

3.3 Sequence patterns and schedulability analysis

In a more general case, the first transition can occur multiple times in the sequence pattern σ whose schedulability has to be investigated. The decorator may

Table 1. Firing times of straight task executions

task/transition	t_0	t_1	t_2	t_3	t_4	t_5	t_6
$t_0t_3t_1t_2t_4t_5t_6$	[0,1]	[2,4]	[3,6]	[2,4]	[4,8]	[4,9]	[4,11]
$t_0t_1t_3t_2t_4t_5t_6$	[0,1]	[1,4]	[2,6]	[2,5]	[3,8]	[3,9]	[3,11]
$t_0t_1t_2t_3t_4t_5t_6$	[0,1]	[1,4]	[2,5]	[2,5]	[3,7]	[3,8]	[3,10]

Table 2. Time span of tasks when $m = 3$ and $n = 2$

task/termination	t_6
$t_0t_3t_1t_2t_4t_5t_6$	$[18,56]$
$t_0t_1t_3t_2t_4t_5t_6$	$[18,56]$
$t_0t_1t_2t_3t_4t_5t_6$	$[18,55]$

find itself in a situation where it has already recognized a prefix of the sequence and now detects an unexpected transition. It may happen that the concatenation of this prefix with the unexpected transition constitutes a sequence that has a suffix which is a prefix of σ . For example, suppose that $\sigma = t_0t_1t_0t_2t_3$ and that the following sequence $\omega = t_0t_1t_0t_1t_0t_2t_3$ occurs. When the decorator automaton meets the second occurrence of t_1 , it knows that the partial sequence $t_0t_1t_0t_1$ is not a prefix of σ , however it must not restart the recognition of σ from scratch because t_0t_1 , which is a suffix of the encountered part of ω , is also a prefix of σ . In this situation the decorator must restart its operation from the point it met the first occurrence of t_1 . This is a common issue in pattern matching problems and efficient algorithms for dealing with these situations are well-known in the literature, e.g. the Knuth-Morris-Pratt algorithm [13]. The decorator may be easily adapted so as to be able to recognize such type of sequences, thus enabling schedulability analysis and the evaluation of the time span of sequences (tasks). When the first transition of a sequence pattern may appear $n > 1$ times, the usage of one clock, which is reset each time the decorator detects the start transition, is not enough for the correct computation of the time span of the sequence. In these cases it is necessary to resort to a clock queue. Each time an occurrence of the first transition is detected, a clock must be reset and put at the end of the queue. When an unexpected transition is encountered, a number of clocks, which is equal to the discarded occurrences of the first transition, must be removed from the head of the queue. At the time the last transition gets matched, the duration of the sequence pattern corresponds to the value of the clock at the head of the queue. Obviously, the maximum length of the queue is equal to n . For this reason the queue can be easily implemented as a circular array of n clocks. From the sequence pattern it is possible to design a non deterministic automaton (NDA) for accepting sequences containing the pattern, which has a number of states equal to the length of the pattern plus one. For example, the NDA depicted in Fig. 6 corresponds to the sequence pattern $\sigma = t_0t_1t_0t_2t_3$.

Although the construction of a deterministic automaton corresponding to a non deterministic one is always possible, in the general case the number of states of the resulting automaton is exponential in the number of states of the original automaton. In the case of NDA arising in sequence-matching, though, the construction of the equivalent deterministic automaton can be done efficiently and the resulting automaton has the same number of states as the original one [14]. Fig. 7 is a TPN model of an assembly subsystem where two raw parts A and B are concurrently processed by Processor 1 and Processor 2, respectively modelled by transitions t_0 and t_2 . The processing of part A is simpler than that

of part B and usually terminates before. After Processor 1 completes its job, the production system waits for a certain amount of time for the part B to be ready. If the processing of part B delays too much, part A is sent again to Processor 1 for a refinement. Part A may be refined a maximum number of times before part B is ready. When both parts are ready, they get assembled together (firing of transition t_3). The timing constraint which governs the refinement of part A is modelled by transition t_1 . Token generation in place p_4 corresponds to the maximum allowed number of refinements of part A.

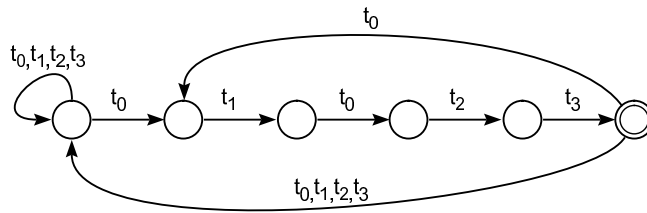


Fig. 6. Non deterministic automaton accepting $\sigma = t_0t_1t_0t_2t_3$

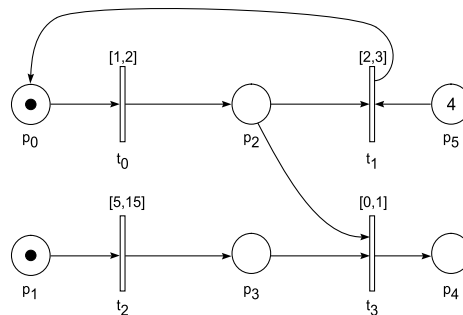


Fig. 7. A FMS/TPN model which admits sequence pattern $\sigma = t_0t_1t_0t_2t_3$

A firing sequence that contains the pattern $\sigma = t_0t_1t_0t_2t_3$ happens when the refinement of part A occurs at least once before the two parts are assembled and the last refinement completes before part B is ready. In order to evaluate the time span of such a sequence, a decorator automaton based on the deterministic automaton corresponding to the NDA of Fig. 6 has to be built. Fig. 8 shows in a simplified way the decorator TA for the problem at hand. Since the first transition (t_0) occurs twice in σ , an array of two clocks must be introduced. As one can see from Fig. 8, an integer variable s is used to store the index corresponding to the head of the clock queue. When transition t_1 is detected and the decorator finds itself in location $d010$, only one clock must be discarded and this is imple-

mented by modular increment of variable s . In all the other locations when the sequence restarts, all the clocks must be discarded.

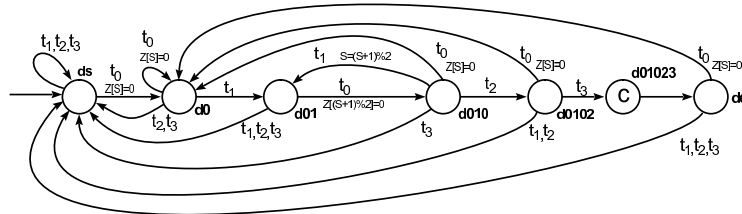


Fig. 8. Decorator for handling sequence $\sigma = t_0t_1t_0t_2t_3$

According to the timing constraints reported in Fig. 7, the sequence pattern $\sigma = t_0t_1t_0t_2t_3$ may not occur. This is witnessed by the fact that the following property is found not satisfied by the model checker:

```
A[] M[4]==1 imply (Decorator.d01023||Decorator.de).
```

However, the following query

```
E<> M[4]==1 && (Decorator.d01023||Decorator.de)
```

returns true mirroring the fact that the pattern σ can really occur. The time span of σ can be computed by respectively finding the maximum value of $k1$ and the minimum value of $k2$ which satisfy the following properties:

```
A[] (M[4]==1 && Decorator.d01023) imply z[s] >= k1
A[] (M[4]==1 && Decorator.d01023) imply z[s] <= k2
```

For the TPN model in Fig. 7 the time span of σ is the interval $[3, 8]$.

4 About the Approach

The approach described in this paper is capable of exploiting the compact data structures and efficient algorithms of UPPAAL. A critical factor is the number of clocks required by a translated TPN model. Indeed, the verification process is exponential in the number of clocks. However, although each transition is associated with a clock, it is the number of simultaneously active clocks which concretely affects space/time complexity of the model checker. In other words, it is the concurrency degree of the TPN model, i.e. the number of simultaneous enabled transitions, which dictates the number of active clocks and then the efficiency of model verification.

The tool combination TPN DESIGNER-UPPAAL was compared to similar combination ROMEO-UPPAAL proposed in [8], so as to figure out some performance index when the tools are applied to “large” models. The same models used in [8] for comparing ROMEO-UPPAAL with the direct use of TINA and ROMEO alone, were experimented with TPN DESIGNER-UPPAAL. Gros1, Gros2 and Gros3 in Table 3 are benchmark models contained in ROMEO distribution v2.6.3.

Particularly for the large models, the approach of this paper outperformed considerably that based on ROMEO-UPPAAL. After an analysis of unexpected not so good behaviour of ROMEO-UPPAAL it soon emerged that a poor management of partial order, i.e. of non determinism during the disabling process of transitions, was the cause of bad performance and not scalability of translation from ROMEO to UPPAAL.

Table 3 reports wall clock time and memory requirements of a few experiments for the three cases: ROMEO-UPPAAL original, ROMEO-UPPAAL modified (partial-order problem fixed by us), TPN DESIGNER-UPPAAL. Performance indexes rely on the memtime facility. All the experiments were carried out on a Linux platform running kernel 2.6.20, on a Pentium IV 3.4GHz, 1GB RAM. For each model, verifyta was launched with the query $A[] \text{ !deadlock}$ whose satisfaction forces the generation of the whole state graph. The production cell model [10] was also chosen because it exhibits a good concurrency degree. The model was simply scaled by replicating the base model.

Table 3. Tools comparison experimental data (UPPAAL version 4.0.6)

	ROMEO-UPPAAL		ROMEO-UPPAAL-mod		TPN DESIGNER-UPPAAL	
	Time (sec.)	Memory (kB)	Time (sec.)	Memory (kB)	Time (sec.)	Memory (kB)
Gros1	363.44	168520	18.89	40848	17.41	40344
Gros2	2297.33	798488	31.96	42016	27.85	41212
Gros3	-	out of memory	46.43	42992	39.70	43040
ProdCell 1 instance	1.98	48460	0.11	2608	0.10	2608
ProdCell 2 instances	-	out of memory	5.48	39448	6.70	39448
ProdCell 3 instances	-	out of memory	1618.92	335108	1579.53	197776

The experiments confirmed that the approach based on TPN DESIGNER-UPPAAL is of practical use. The tool behaves similarly to ROMEO-UPPAAL modified but seems to scale up a little bit better. All of this can depend on the fact that broadcast synchronizations during completion of a transition firing are embedded in the automaton in Fig. 3 whereas they are part of a separate SUPERVISOR automaton in [8]. Scenarios where both tools are difficult to apply are in any case out of scope of both TINA and ROMEO which rest on “off-line” generation of the time reachability graph. For very complex models, where exhaustive verification is almost impossible, simulation techniques like those described in [10] can be applied to estimate timing properties.

5 Conclusions

This paper proposes an original approach to schedulability analysis of real-time systems specified by Time Petri Nets (TPNs). The approach is based on a translation of a TPN model on to the popular UPPAAL tool, assisted by the TPN DESIGNER toolbox [7]. The approach, similarly to the method described in [22], permits one to investigate behavioural and timing properties of transition sequences (execution tasks). In addition, this paper shows how the approach can be generalized to the handling of sequence patterns in which the first transition can occur multiple times and where (possibly) the recognition process requires to be restarted while ensuring timing analysis. On-going work is geared at

- extending TPN DESIGNER so as to offer TCTL queries directly at the TPN level, thus allowing the modeller to reason in TPN terms and hiding the underlying use of the UPPAAL model checker engine
- applying the approach to the timing analysis of event scenarios in real-time actor systems, e.g. [11]. The idea is to model message exchanges of an actor computation by TPNs and then to investigate event timing constraints so as to guide the runtime behaviour of the scheduler.

References

1. R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proc. of Fifth Annual IEEE Symposium on Logic in Computer Science (LICS'90)*, pages 414–425, 4-7 June 1990.
2. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, Apr. 1994.
3. G. Behrmann, A. David, and K. G. Larsen. *Formal Methods for the Design of Real-Time Systems*, chapter A Tutorial on UPPAAL, pages 200–236. Number 3185 in LNCS. Springer, 2004.
4. 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.
5. B. Berthomieu, P.-O. Ribet, and F. Vernadat. The tool TINA—Construction of abstract state spaces for petri nets and time Petri nets. *International Journal of Production Research*, 42(16):2741–2756, 2004.
6. G. Bucci and E. Vicario. Compositional validation of time-critical systems using communicating time Petri nets. *IEEE Transactions on Software Engineering*, 21(12):969–992, Dec. 1995.
7. L. Carullo, A. Furfaro, L. Nigro, and F. Pupo. Modelling and simulation of complex systems using TPN Designer. *Simulation Modelling Practice and Theory*, 11(7-8):503–532, Dec. 2003.
8. F. Cassez and O. H. Roux. Structural translation from time Petri nets to timed automata. *Journal of Systems and Software*, 79(10):1456–1468, Oct. 2006.
9. F. Cicirelli, A. Furfaro, and L. Nigro. An approach to protocol modeling and validation. In *Proc. of 39th Annual Simulation Symposium (ANSS'06)*, pages 261–268, 2-6 April 2006.

10. F. Cicirelli, A. Furfaro, and L. Nigro. Distributed simulation of modular time Petri nets: An approach and a case study exploiting temporal uncertainty. *Real-Time Systems*, 35(2):153–179, Feb. 2007.
11. G. Fortino, L. Nigro, F. Pupo, and D. Spezzano. Super actors for real time. In *Proc. of 6th IEEE CS Workshop on Object-oriented Real-Time Dependable Systems (WORDS'01)*, pages 142–149, 2001.
12. G. Gardey, D. Lime, M. Magnin, and O. H. Roux. Romeo: A tool for analyzing time Petri nets. In *Proc. of 17th Int. Conf. on Computer Aided Verification (CAV'05)*, number 3576 in LNCS, pages 418–423. Springer, 2005.
13. D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.
14. H. R. Lewis and C. H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
15. M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear controller. *International Journal on Software Tools for Technology Transfer (STTT)*, 3(3):353–368, Aug. 2001.
16. P. Merlin and D. Farber. Recoverability of communication protocols—implications of a theoretical study. *IEEE Transactions on Communications*, 24(9):1036–1043, September 1976.
17. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
18. C. Ramchandani. *Analysis of asynchronous concurrent systems by timed Petri nets*. PhD thesis, MIT, Cambridge, MA, 1974.
19. ROMEO - A tool for time Petri net analysis. <http://romeo.rts-software.org/>.
20. TINA - TIme petri Net Analyzer. <http://www.laas.fr/tina/>.
21. UPPAAL. <http://www.uppaal.com>.
22. D. Xu, X. He, and Y. Deng. Compositional schedulability analysis of real-time systems using time Petri nets. *IEEE Transactions on Software Engineering*, 28(10):984–996, Oct. 2002.