Minotor: Monitoring Timing and Behavioral Properties for Dependable Distributed Systems

Olivier Baldellon, Jean-Charles Fabre, Matthieu Roy

MINOTOR: Monitoring Timing and Behavioral Properties for Dependable Distributed Systems

Olivier Baldellon*‡, Jean-Charles Fabre*‡, Matthieu Roy*†
* *CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France*
† *Univ de Toulouse, LAAS, F-31400 Toulouse, France*
‡ *Univ de Toulouse, INP, LAAS, F-31400 Toulouse, France*

***Abstract*—Assessing the correct behavior of a given system at run-time can be achieved by monitoring its execution, and is complementary to off-line analysis such as static verification.**

In this work, we focus on run-time monitoring of system properties that include both causality and timing constraints, in distributed and time-constrained systems. Based on a description of a property that includes events and temporal constraints, expressed as a timed-arc Petri net, we show how to automatically transform it into a an executable and distributed monitoring engine.

To that aim, we introduce a modification of the semantics of Petri nets to be able to execute it online on partial executions and distributed observation environments. We show how to use this formal framework to provide MINOTOR, a model-driven distributed monitoring system, describe its implementation and show its applicability on a transportation use-case.

***Keywords*-Distributed Systems; Time-constrained Systems; Online Monitoring; Fault-tolerant Systems; Petri nets.**

INTRODUCTION

Supervising or monitoring applications states is a requisite to detect a possible violation of system specification and envisage a recovery action. Alas, on-line monitoring of complex, distributed and real-time systems is a highly complex task that, to our knowledge, has not yet been fully tackled. On-line monitoring of applications, be them distributed or not, in a centralized way is an active research domain (see, e.g., [MR10], [JRR94], [BLS06], [RFR08], [ZSLL09]), but, as we will see in Section I, these solutions do not provide a distributed implementation of monitors that can handle both distributed and real-time specifications.

Most formalisms available to express event-based behavioral properties are inherently centralized, partly because they were developed to model check the system before its deployment, a step that does not require a distribution of the verification process. In our approach, we want to verify an application in operation, and thus distributing the monitoring process is a crucial issue. The first step towards the distribution of monitoring is to use a formalism in which the state of the system is distributed.

Following this reasoning, timed-arc Petri nets are good candidates for modeling a distributed embedded system with timing constraints, since this formalism is decentralized by nature, and allows to express *local* timing constraints. The dual nature of the formalism thus allows to reason both on temporal and behavioral properties that may be distributed among a set of nodes.

In this paper, we focus on the run-time monitoring of system properties expressed as timed-arc Petri nets. More precisely, the monitoring system uses timed events to trigger transitions in a Petri net that describes behavioral and temporal properties of the system. We assume a global clock enables events to be time-stamped.

The structure our approach is schematized in Figure 1. Intuitively, the system under supervision raises timed events — i.e., couples (e_i, τ_i) meaning that event e_i happened at time τ_i — which are then captured by the monitor system that triggers the firing of transitions in the Petri net that models behavioral and temporal properties of the system.

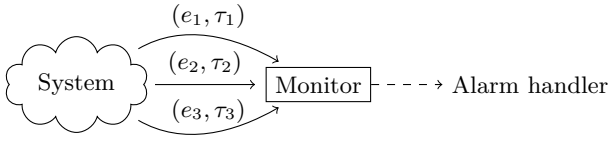


Figure 1. General approach

We aim at detecting violation of system properties:

1) events not generated by the actual distributed system in operation (missing events);
2) wrong order of raised event (event e_2 appears before e_1 whereas e_2 should have happened after e_1);
3) deadline missed by some event.

The detection of the above properties should be performed as soon as possible, for obvious detection latency reasons, without requiring a perfect observation: due to the distribution of generated events, and the time required to dispatch events to the monitoring

This work has been partially supported by ANR French national program MURPHY (grant #ANR-10-BLAN-0306).

system, events may be received in an order that does not correspond to their real-time occurrence.

We assume that the specification (i.e., correct behavior) of the monitored application is provided as a Petri net. In a classical *weak* Petri net semantics [BR08], the firing of a given transition results in the removal of tokens in input places and the creation of others in output places. Thus, a transition is fireable if and only if the tokens are already present in the input places.

Intuitively, we propose to fire a transition as soon as the corresponding system event is received, no matter if tokens are present in input places or not. We introduce the concept of *negative* tokens to mark the fact that a transition was fired, independently from the marking of input places. Allowing negative tokens permits us to fire a transition associated to an event *as soon as the event is captured* without blocking the execution of the model.

The main benefit of this approach is that transitions can be fired independently as soon as the corresponding system event appears. This has two major interests: firstly, it allows to perform most of computation locally, reducing required communication and synchronization between processes. Secondly, it allows to completely distribute the monitoring process and to optimize it based on system topology, enabling a better resources utilization.

Summary of contributions.: In this paper, we describe in details a monitoring strategy that was briefly presented in [BFR12]. Based on a modification of Petri net semantics, we provide algorithms that implement efficiently this strategy, and develop a thorough analysis of timing hypotheses required to implement our monitoring framework. We provide measurements on automatically generated patterns to test the scalability of the approach, and model, for proof-of-concept, a simplified automated railway transportation scenario.

In a first part, we compare our approach with other related techniques of on-line monitoring. Section II briefly presents the formalism used, and in particular the extension of Petri nets to include negative tokens. Section III describes a protocol to distribute the monitoring engine, based both on the Petri net and on physical mapping considerations. Section IV describes our implementation. Section V exemplifies the applicability of the approach.

I. State of the art

Most previous works on monitoring of real-time properties are based on centralized monitoring of properties expressed using Linear Temporal Logic (LTL) formulas or Timed Automata [MR10], [BLS06], [RFR08]. Unfortunately, both logic formulas and timed automata are inherently centralized and cannot be easily adapted for distributed monitoring. Usually, an LTL formula is mapped to its corresponding timed automata equivalent for run-time evaluation. A timed automata has an instantaneous state that has no natural decomposition in sub-states and thus cannot be easily distributed.

Other approaches explicitly target the distribution of the monitoring process [ZSLL09], [JRR94]. [JRR94] allows a distributed monitoring of properties that are expressed as conjunction of simple logic properties, while [ZSLL09] proposes, in some cases, the distribution of properties expressed in the temporal logic based MEDL specification language. Yet, in both approaches, the formalism proposed is not as expressive as timed-arc Petri nets, that can express complex and, more importantly, distributed behaviors.

Centralized monitoring of asynchronous [FBHJ05] or real-time [CJ05] distributed systems using Petri nets has also been studied. In the above works, the main challenge solved consists in finding, from an observed events sequence, a corresponding Petri net execution to "explain" the observation. The aim we are pursuing here is different: to simplify the event/transition correspondence, we require that an event corresponds to exactly one transition. Moreover, we are interested in distributed monitoring of systems, and thus we need to accommodate with delayed observations, missing events and faulty behaviors.

In the distributable nets approach [Hop91], one executes a Petri net in a distributed network, following a *location function* such that if a place is an input of a transition, then this place and this transition must be on the same node — that means that the decision of firing a transition can be done locally. Although distributable nets are not directly related to the work presented here, we adapted the location function concept while overcoming the "same node" requirement, in order to be able to fire transitions as soon as events are caught.

II. Petri Nets For System Properties

As mentioned in the introduction, our monitor tool, Minotor, is a distributed service that executes a model of the system expressing temporal and behavioral properties on reception of timed events from the system, *cf.* Figure 1. In this section, we first recall some classical definitions on timed-arc Petri nets, before describing informally its new semantics. Finally, the generalization of timed events is presented and discussed.

A. Timed-Arc Petri Nets

A timed-arc Petri net is a tuple $(P, T, {}^\bullet(\cdot), (\cdot)^\bullet, M_0, I)$, where P is a finite set of *places*, T is a finite set of *transitions*, M_0 is the initial marking (a function that associates to each place a set of tokens), ${}^\bullet(\cdot)$ (respectively $(\cdot)^\bullet$) is the *backward* (resp. *forward*) incidence function that associates to each transition the set of places from which the transition starts (resp. to which the transition ends), and I a *time constraint function* that associates

a time interval to every arc from a place to a transition. We say that there is an arc from a place p to a transition t if $p \in {}^\bullet t$.

We assume there is a bijection between transitions and events generated by the system. Thus a timed event (e_i, τ_i) can always be written (t_i, τ_i) where t_i is the transition associated to e_i.

B. Petri Net Execution

As explained in the previous part, the properties on system events that need to be monitored are expressed using the timed-arc Petri net formalism. Our approach consists in executing a Petri net on-the-fly to detect failures. The monitoring tool takes as an input a sequence of events, i.e., a sequence of couples (t, τ) where t represents the transition associated to an event and τ the date of the event. The monitoring tool executes the model using such events sequence, and possibly raises an alarm when it detects an incorrect behavior.

A simple execution: Let us take as an example the simple Petri net depicted in Figure 2: if the monitor receives, in this order, the sequence of events $(t_1, 10); (t_2, 15); (t_3, 21)$, it will first fire t_1 and remember that this firing was done at time 10. When the event corresponding to t_2 is received, the transition t_2 is fired because $\tau_2 - \tau_1 = 15 - 10 \in [3, 6]$. The reception of $(t_3, 21)$ will raise an error because the event occurred too late; the token stayed in the place of ${}^\bullet t_3$ six units of time ($\tau_3 - \tau_2 = 21 - 15 = 6$), which is out of the specified interval $[0, 5]$. However, as the corresponding event happened, the transition t_3 will still be fired to enable the execution of the Petri net model to continue, and an error will be reported.

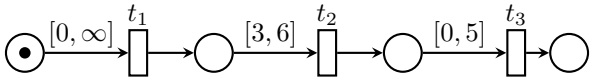


Figure 2. A simple Petri net

In this scenario, notice that the minimal amount of information to detect a failure is $(t_2, 15); (t_3, 21)$: whatever the occurrence time of t_1, t_3 happened too late with respect to t_2. If the monitor only receives $(t_2, 15); (t_3, 21)$ but $(t_1, 10)$ is not yet received, the usual semantics of Petri nets forbids to fire the transitions t_2 and t_3; in other words, to fire t_2 and consequently t_3, the transition t_1 needs to have been already fired. This crucial issue can be solved using negative tokens, as shown below.

An execution with negative tokens: Our approach consists in firing a transition t as soon as it is received and anticipating the removal of tokens in places of ${}^\bullet t$ by "adding" negative tokens in these places. Figure 3 shows the result of firing t_2, with a negative token represented as a black circle, a positive token as a black disk.

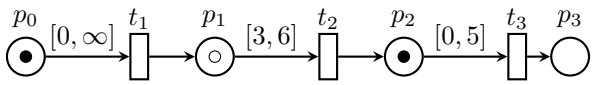


Figure 3. After the firing of t_2

The firing of t_3 will add one negative token in p_2 and one positive token in p_3, as shown in Figure 4. The removal of a positive token always depends on the presence of its negative counterpart. To know how long the positive token stayed in the place p_2, we need to compare the date of the event that created the negative token with the date of the event that created the positive one. If the difference is not in the time interval $I(p_2, t_3) = [0, 5]$, then an error is raised. In all cases, both tokens are removed.

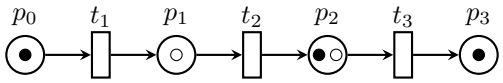


Figure 4. After the firing of t_3

In summary, the main difference between our approach and classical semantics is that, in the classical one, a transition is allowed to be fired only if it all required tokens are present. In our approach, transitions are always fired speculatively, and the fireability property is checked *a posteriori*.

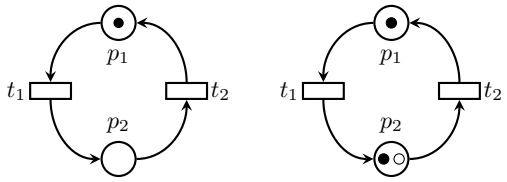
C. Tags

As classical Petri nets tokens are extended to include both positive and negative tokens, this implies that the monitoring system has to add information to be able to know unambiguously to which negative token a positive one may be associated to. In particular, the problem of having to distinguish between tokens may appear when a given transition is fired several times in the same execution, at different dates, within a cycle.

As an example, let us consider the Petri net of Figure 5(a) and the following execution (t_1, τ_1); (t_2, τ_2); (t_1, τ_3); (t_2, τ_4) in which the token moves to p_2, goes back to p_1, moves again to p_2 and finally returns in p_1. Let us suppose that events (t_2, τ_2) and (t_1, τ_3) were not received. The observed execution is then $(t_1, \tau_1); (t_2, \tau_4)$ and the resulting Petri net state is the one of Figure 5(b).

The missing event (t_2, τ_2) would have created a negative token in p_2, corresponding to the positive one generated by (t_1, τ_1). Similarly, the missing event (t_1, τ_3) would have created a positive token corresponding to the negative one generated by (t_2, τ_4). Consequently, the positive and negative tokens present in place p_2 do not

match: the positive token was produced by (t_1, τ_1) while the negative one was created by (t_2, τ_4).



(a) Beginning of Execution (b) With missing events

Figure 5. Token and tags

To give a more concrete example, let us suppose that t_1 corresponds to the beginning of an action and t_2 to its end. The value $\tau_2 - \tau_1$ corresponds to the time taken by the first occurrence of this action and $\tau_4 - \tau_3$ is the time taken by the second occurrence execution; however, the value $\tau_4 - \tau_1$ represents the time taken by two occurrences, and thus does not correspond to timing constraints expressed in the specification.

The issue to answer is "given a positive and a negative token, do those two tokens match?". To address this issue, we introduce the notion of *tags*. A tag is a unique identifier added to tokens. A positive and a negative tokens are related if and only if they have the same tags (cf. Figure 6). The main difference with colored tokens is the following: colored tokens are used to extent the expressivity of a Petri net while tagged tokens are only used to allows an out-of-order execution of a given Petri net.

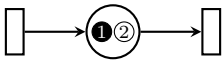


Figure 6. Two tokens that cannot match

As a consequence, two nodes of the monitor in charge of generating tokens for a same place need to agree on tag generation for the tokens of the place. If agreement is not possible, then it is impossible to monitor properties in presence of missing events without false negatives, i.e., undetected errors may occur. One can easily notice that being able to compute the running time of a task requires to unambiguously associate the two events corresponding to its beginning and its end.

D. Events

Until now, events were described as couples (t, τ) where t is a transition and τ a date. Now we need to refine this definition to take into account tags.

Since tags are added at event generation, a tagged event is a tuple (t, τ, f) where t is a transition, τ a date and f the tag function defined on ${}^\bullet t \cup t^\bullet$. The function f associates to each place of ${}^\bullet t \cup t^\bullet$ a tag. Then, at the reception of the event (t, τ, f), a positive token, tagged $f(p)$, is added in every place p of $t^\bullet$ and a negative token, tagged $f(p)$, is added in every place p of ${}^\bullet t$.

On the example of Figure 5, a simple implementation of f is to tag each token by the occurrence number of the corresponding action (i.e., the first firing of t_1 and t_2 is tagged 1, the second occurrence is tagged 2, etc.)

III. Monitoring with Negative Tokens

The first part of this section describes the protocol main ideas, and provides a relatively straightforward transcription of the execution semantics described previously with negative tokens. A second part deals with the timing aspects of the monitoring engine: given a fully synchronous communication system, we show how to compute the timeouts for handling deadline misses and missing events. Then, we dig into the physical mapping of the monitoring engine on a distributed set of nodes.

A. Protocol description

The protocol assigns a monitoring thread to each place and each transition of the Petri net. When the system generates an event, this event is sent to the thread associated to the corresponding transition. In return, transition threads send tokens to threads associated to places when they receive events and fire transitions.

The transition thread for transition t is described in Algorithm 1; notation "p ! msg" means that message msg is sent to the thread associated to place p, and Pre and $Post$ variables refer to the two sets ${}^\bullet t$ and $t^\bullet$.

Algorithm 1 Thread associated to transition t

```
1: procedure TRANSITION(Pre,Post)
2:     When an event e = (t, τ, f) is received do
3:         for p ∈ Pre do p ! (−, t, τ, f(p))
4:         end for
5:         for p ∈ Post do p ! (+, t, τ, f(p))
6:         end for
7:     done
8: end procedure
```

The place thread is described in Algorithm 2. When a thread associated to a place receives a token, it checks if it already received the negative version of this token (same tag, but opposite sign). In this case, the place thread needs to compute the difference between the two dates and to compare it with the corresponding timing interval given by the I function before deleting the two corresponding tokens. The detection of the violation of temporal properties is partially done during this comparison: the comparison of the two dates may raise an error if timing assumptions are violated.

If no opposite sign token has been received before, the token is stored and a new timer is started, to be able to detect a deadline miss of the opposite sign

token. The timer is required in addition to the difference and comparison mechanism described above: if the opposite counterpart of a token is never received, then the comparison mechanism will not be triggered. To be able to trigger an alarm for a missing event, this timeout mechanism is implemented in the timer function described in Section III-B. Interestingly, this mechanism allows also for the detection of deadline misses as soon as they are detectable.

The main goal of the SetTimer function is to detect the fact that an event did not arrive on time. The exact computation of the value of the timeout is described in the next section. The timer runs in a separate thread that sends to the place a message *timeout*(*token*) when the timer expires. If a place receives such message when the corresponding token is still in memory, tokens are erased, and a timing error is raised up.

Algorithm 2 Place thread associated to p

```
1:  procedure PLACE(I)
2:      When token = (sign, t, τ, tag) is received do
3:          if ∃token' = (−sign, t', τ', tag) ∈ Mem then
4:              Mem ← Mem−token'
5:              CancelTimer(token')
6:              if sign = +
7:                then ok = ((τ' − τ) ∈ I(p, t'))
8:                else ok = ((τ − τ') ∈ I(p, t))
9:              end
10:             if ¬ok then RAISE(timing failure)
11:         else
12:             Mem ← Mem+token
13:             SetTimer(token,t,p)
14:         end if
15:     done

16:     When timeout(token) is received do
17:         Mem ← Mem−token
18:         RAISE(timing failure)
19:     done
20: end procedure
```

B. Timers

The very existence of timeouts is conditioned by the ability to bound both message transfer delays and reaction delays. In this part, we explore timer computation in an *optimistic* fashion —we want a timer to expire only when there is *for sure* a timing error in the system—, since most timing verifications will be handled by the difference and comparison mechanism (lines 7–8 of Algorithm 2).

For this purpose, we need to introduce additional assumptions on the communication layer, both for the actual system in operation and for the monitoring system. Notice that the existence of such bounds seems natural in any system that implements real-time mechanisms with timeouts in the specification, which is the core target of our approach.

We assume that there are bounds on message transfer delays in the whole system. More precisely, we assume that a message sent by a thread th at time τ will be received by time $\tau+\Delta_{th,th'}$ by thread th'. We also assume the existence of a second function δ that gives, for any transition, the delay needed for an event to be received by the thread associated to a transition. If an event happens in the system at time τ, then the corresponding transition t will receive this event before time $\tau + \delta(t)$.

The existence of those two functions Δ and δ implies that all links in the system are synchronous. Indeed, Δ and δ exactly represent the time needed to route a message between the different participants of the monitoring system.

Now we dig into timers computations; when a token is received by a place thread in Algorithm 2, and no corresponding opposite signed token exists, a timer is set (line 13) and, would the timer expire, it would raise a timing failure. Notice that all computations here are *optimistic*, i.e., we want a timer to expire only when there is *for sure* a timing error in the system.

Let us consider a place p in which a transition t_1 creates a positive token at time τ_1, and t_2 produces its negated version at time τ_2 (for example, Figure 5 with place p_2). Now if the arc to the transition t_2 is time constrained by $I(p, t_2) = [a, b]$, then by definition of the semantics,

$$a \leq \tau_2 - \tau_1 \leq b \quad (*)$$

A positive token is received first: In this case, the worst case corresponds to an immediate token transfer, i.e., the positive token is received as soon as event corresponding to t_1 is fired (see Figure 7).

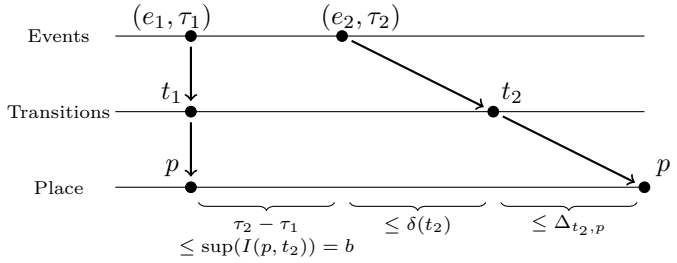


Figure 7. Timer computation : positive token is received first

For any transition t_2 in $p^\bullet$, the worst case corresponds to an event fired at time $\tau_1 + b$, by the above inequality $(*)$. Now for transfer delay times, the worst case is $\delta(t_2) + \Delta_{t_2,p}$, and hence, place p has to receive the corresponding negative token by time $\tau_1 + \sup(I(p, t_2)) + \delta(t_2) + \Delta_{t_2,p}$ (cf. Figure 7).

As place p does not know which transition of $p^\bullet$ will create the negative token, the same reasoning applies to all transitions of $p^\bullet$, and we can finally deduce that p

needs to wait until $\tau_1 + T_+(p)$ with:

$$T_+(p) = \max_{t \in p^\bullet} \{\Delta_{t,p} + \delta(t) + \sup(I(p,t))\} \qquad (1)$$

A negative token is received first: As in the previous case, the worst case corresponds to an immediate token transfer, i.e., the negative token is received as soon as the event corresponding to t_2 is fired (see Figure 8).

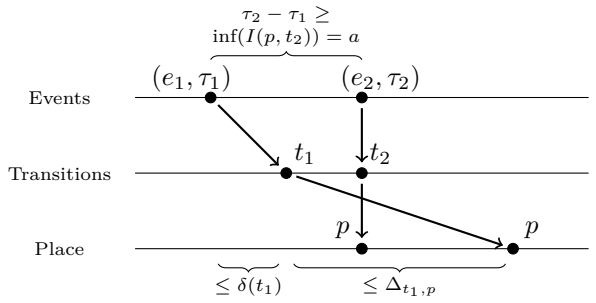


Figure 8. Timer computation: negative token is received first

Using inequality $(*)$, it is easy to check that (e_1, τ_1) happens at most at time $\tau_2 - \inf(I(p,t_2)) = \tau_2 - a$. Now, considering worst case communication delays, p receives the token associated to e_1 at most at time $\tau_2 - \inf(I(p,t_2)) + \delta(t_1) + \Delta_{t_1,p}$ (cf. Figure 8). Contrarily to the previous case, here p knows which transition of $p^\bullet$ to wait for, and hence p arms a timer that expires at $\tau_2 + T_-(p,t_2)$ with:

$$T_-(p,t_2) = \max_{t \in {}^\bullet p}\{\Delta_{t,p} + \delta(t)\} - \inf(I(p,t_2)) \qquad (2)$$

The description of the SetTimer algorithm used by the place thread of Algorithm 2 to arm watchdogs when waiting events is described in Algorithm 3.

Algorithm 3 SetTimer procedure

```
1: procedure SetTimer(token,t0,p)
2:     if token.sign > 0 then
3:         T = max_{t∈p•}{Δ_{t,p} + δ_t + sup(I_{p,t})}
4:     else
5:         T = max_{t∈•p}{Δ_{t,p} + δ_t} − inf(I_{p,t0})
6:     end if
7:     wait T
8:     p ! timeout(token)
9: end procedure
```

C. Physical mapping

An important question that needs to be addressed now concerns the physical mapping of the threads of the protocol. Intuitively, an adequate placement strategy would locate the transition thread as close as possible to the node where the corresponding event is generated.

In the case a transition thread is located on the node that generates the event of its transition, the failure of the node will result in the failure of the place thread. This case is not problematic: the failure of the transition thread th_t will result in a timeout for associated tokens in place threads that depends on transition t. Such timeouts will raise timing failures (see Algorithm 2) : the failure will be detected.

Contrarily, if a place thread is collocated to transition threads it depends on, the crash of the node will not be detected, since only place threads may raise errors. Indeed, the optimal placement of place threads is the result of a trade-off between low detection latency and fault tolerance.

In order to attain a low latency detection while preserving fault tolerance properties, and mitigate the above mentioned trade-off, we use a classical replication mechanism. In this strategy, place threads are duplicated on several node, and events are sent to the set of replicas using a uniform reliable broadcast service. Let us suppose that the thread associated to a place p is replicated on k nodes. To maintain consistency between replicas, we want the k replicas receive the same set of tokens. Notice that, since our approach allows out-of-order reception of tokens, the required service is a simple Uniform Reliable Broadcast [Ray10], thus does not require costly consensus-based Atomic Broadcast.

IV. Implementation

We have developed Minotor[1], an implementation of the approach in the Erlang language. The implementation heavily uses multithreading, assigning one lightweight thread per place and one per transition. Indeed, the implementation has proved to be very scalable, since we were able to run series of tests on Petri nets of size up to 2^{20} (more than one million) transitions and places.

A. Timing hypotheses

We assumed we have an upper bound D of $\Delta_{t,p} + \delta(t)$ for every place p and transition t. Intuitively, this means than when an event occurs in the system at time τ, the corresponding tokens will be received by the places before $\tau + D$. Equations 1 and 2 of Section III-B become:

$$T_+(p) \leq D + \max_{t \in p^\bullet} \{\sup(I(p,t))\}$$
$$T_-(p,t_2) \leq D - \inf(I(p,t_2))$$

To compute the two above values to set timers, place p needs to know the intervals $I(p,t)$ for transitions t of $p^\bullet$. Thanks to this simplification, in the implementation a place p needs to be aware of transitions in $p^\bullet$ only, but not of the ones in ${}^\bullet p$.

[1] http://www.olivier.baldellon.eu/documents/minotor-prdc.tar.gz

B. Petri net deployment

To create an arc, a message is sent to the target transition thread. When the arc concerns an input place, for instance `{arc, "p1", "t1", 0, infinity}`, the transition thread `t1` sends to the place thread `p1` the interval $[0, +\infty]$. The processing of this message will be described in the next section, when explaining the transition thread code. When the arc concerns an output place, as `{arc, "t1", "p2" }`, no message needs to be sent to places $t^\bullet$, due to the simplification explained above.

The deployment philosophy is to be as dynamic as possible: places, transitions and arcs can be added at any time to be able to update the model at runtime.

C. Transition thread in action

The code of the transition thread, presented in Listing 1, is a direct transcription of the algorithm described in Section III, with some additional machinery required to create and deploy dynamically the Petri net.

```
1  -module(transition).
2
3  loop(Name,Pre,Post) ->
4      receive
5          {event,Tags,Time} ->
6              send_token(Pre,minus,Tags,Time,Name),
7              send_token(Post,plus,Tags,Time,Name),
8              loop(Name,Pre,Post) ;
9          {new_arc,Place,Pid} ->
10             loop(Name,Pre,[{Place,Pid}|Post]);
11         {new_arc,Place,Pid,Min,Max}->
12             Pid ! {new_arc,Name,Min,Max},
13             loop(Name,[{Place,Pid}|Pre],Post)
14     end.
```

Listing 1. Code for transitions

In brief, the function `loop` has three parameters: the transition name and two lists, one representing the input places set called `Pre`, and the other one representing the output places set called `Post`. When the function is first launched, the two sets are initialized to the empty list. The loop function waits for the reception of messages, does some action and calls itself again with updated parameters. This is the common way of implementing threads in Erlang, definitely recursive.

Three types of messages can be received: an "event" message and two "new arc" messages. "Event" messages correspond to monitoring messages, while "new arc" messages correspond to deployment operations.

Deployment: When a transition t is asked to create a new arc with a place, it just needs to call recursively the loop function with updated parameters `Pre` and `Post`. For example, line 10 (`loop(Name,Pre,[Place,Pid|Post])`) corresponds to the creation of a new arc between place `Place` and the current transition. The `Post` set is updated with a couple containing both the name `Place` and the address `Pid` of the place. In the case of a new place in $t^\bullet$, then the value of the interval $I(p,t)$ is sent to the place using the `Pid` value: `Pid ! new_arc,Name,Min,Max` (line 12).

Monitoring: When an event is received, tokens must be sent to the two sets `Pre` and `Post`. Negative tokens are sent to `Pre` by means of the statement: `send_token(Pre,minus,Tags,Time,Name)`, line 6. Positive tokens are sent to `Post` by means of the statement: `send_token(Post,plus,Tags,Time,Name)`, line 7. Finally, the loop function is called back with the same parameters: `loop(Name,Pre,Post)` at line 8.

```
start(Timeout,Pid,{plus,Tag,Tp}) ->                          1
   receive                                                    2
     {token,{minus,Tag,Tm},{Min,Max}} ->                      3
        case timer:now_diff(Tm,Tp) of                         4
           D when D > Max  ->                                 5
              Pid ! {too_late,Tag,D};                         6
           D when D < Min ->                                  7
              Pid ! {too_early,Tag,D};                        8
           D ->                                               9
              Pid ! {ok,Tag,D}                                10
         end                                                  11
    after Timeout ->                                          12
          Pid ! {plus_timeout,Tag}                            13
    end.                                                      14
start(Timeout,Pid,{minus,Tag,Tm},{Min,Max}) ->                15
    receive                                                   16
       {token,{plus,Tag,Tp} } ->                              17
          case timer:now_diff(Tm,Tp) of                       18
             D when D > Max  ->                               19
                Pid ! {too_late,Tag,D};                       20
             D when D < Min ->                                21
                Pid ! {too_early,Tag  ,D};                    22
             D -> Pid ! {ok,Tag,D}                            23
          end                                                 24
    after Timeout ->                                          25
          Pid ! {minus_timeout,Tag}                           26
    end.                                                      27
```

Listing 2. Code for timers

D. Places and timers

We do not give in this paper the full implementation of place thread, but rather focus on timing constraint verification. In Listing 2, we show the current implementation of the verification of timing constraints using timers threads.

When a positive token is received by a place p, the place starts a new timer with the parameters `start(Timeout,PlaceId,plus,Tag,Tp)`. The first parameter, `Timeout` corresponds to the value $T_+(p)$, the second one corresponds to the address of p and the last one corresponds to the token with the sign (here `plus`), the `Tag` and the creation date `Tp`. The timer thread waits for the reception of the negative token within the corresponding timing interval [`Min`, `Max`].

Depending on the value `Tm` − `Tp`, i.e. the time spent by the token in the place (line 4), the timer will inform the place of the result. It can be either `ok` (line 10), `too_late` (line 6) or `too_early` (line 8). If the negative token is not received before the timer expires, then the place is

informed by a message `PlaceId ! plus_timeout,Tag` (line 13).

Similarly, the handling of timers for negative tokens is described lines 15–27.

E. Performance and scalability

To test the scalability of our approach, we conducted tests on a simple square Petri net described in Figure 9. This Petri net consists of n concurrent sub Petri nets (n "lines"), every one being a sequence of n actions. We generated the associated Petri nets for $n = 2^k$ with $k = 0..10$ and conducted the experiments on a 8-core HP machine running Debian GNU/Linux (2.89 GHz with 8 GB of memory). The monitor was controlled from an other computer where events were generated. Notice that for $k = 10$ there are 1 048 576 places and 1 049 600 transitions and thus about 2.1 million threads. The memory footprint was measured with Erlang VM running on a single core.

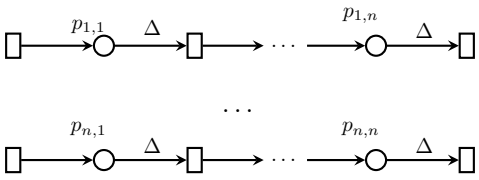


Figure 9. Test Petri net

The table in Figure 10 shows the memory footprint with respect to the value of n. We show that for big enough values of n there is a proportional relation between the number of threads and the payload. The payload corresponds to the difference between the VM memory footprint after the creation and deployment of the Petri net with the initial VM footprint.

n	1 . . . 32	64	128	256	512	1024
Mem. (MB)	410	417	435	746	1731	5664
Payload (MB)	0	7	25	336	1321	5254
#Threads/Payload $(2n+1)n$/Payload	-	-	-	390	397	399

Figure 10. Memory footprint for large Petri nets (table)

For small Petri nets, memory footprint is always 410 MB. This value corresponds to the size of the virtual machine. This value is relatively big because the virtual machine was configured for all tests to be able to run large Petri nets (more than 2 millions of threads), In practice, for small Petri net ($n = 16$, i.e. 496 transitions or places), a virtual machine of 30MB has proved to be sufficient.

Finally, in term of CPU time, the deployment of the Petri net is a costly operation (about 40s to generate the 2 millions thread Petri net). However, the execution, i.e. the processing of received events, is negligible on each node.

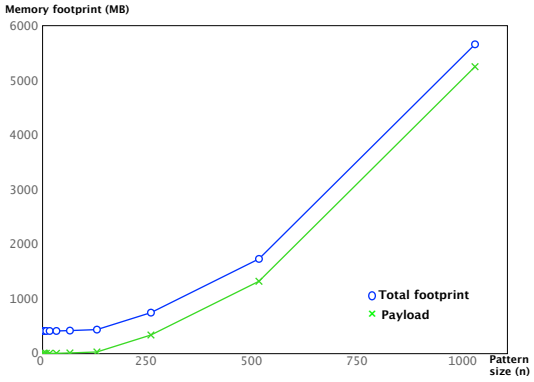


Figure 11. Memory footprint for large Petri nets (graph)

To conclude, the Erlang implementation was a successful proof of concept in terms of feasibility and performance. A more significant implementation of MINOTOR in the Xenomai real-time system as a kernel driver is being developed (work in progress)

V. DERIVING MODELS FROM A CONCRETE EXAMPLE

The monitoring system presented in this paper needs, as an input, a description of the time- and event-based specification of correct behaviors in terms of an timed-arc Petri net. This step has not been fully addressed yet, and, indeed, devising a correct specification as a Petri net is usually not considered to be an easy task.

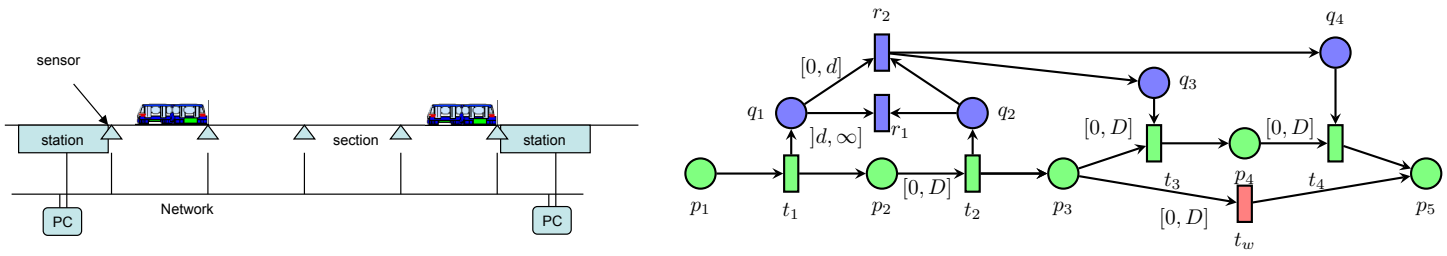
Fortunately, when considering critical systems, most designers provide clear specification of interactions within the system, in order to run unit tests, and to use as an input for model-checking interactions between components. Such specifications may be provided as linear temporal logic formulas, timed automata, state charts, etc. Then, one can use one of the possible automated translations from a given formalism to a Petri net, e.g., [Srb05], [RPC02].

Hence, we assume that we can reuse a description that has already been provided to developers as early as during the testing phase of the system.

To motivate our approach, we here consider a railway transportation supervision system, and show how a model of its behavior can be developed incrementally from its simplest form up to a complex model that encompasses advanced techniques such as nominal and degraded modes of operation.

A. Railway supervision system: overview

In the introduction of this paper we advocated that our approach was able to detect the violation of system properties without blocking despite wrong order of events and deadline misses by some events. In real systems, detection is not enough and thus we assume that the nominal behavior of a given system is complemented



(a) A fully-automated train supervision system

(b) Handling warning states and missing events

Figure 12. Railway supervision system

by at least one alternative behavior, targeting possibly several degraded modes of operation.

In this last part, we briefly illustrate this problem by means of an example. We consider a fully-automated train control system made up of a network of lines, each composed of a set of railway track sections, as depicted in Figure 12(a). Control of a train is supervised from a central command and control system but the trains have some autonomic behavior, i.e. automatic train driving is carried out simply by assigning each train a "target station". In other words, each train starts from an initial station and has a target destination, with several intermediate stations in between. Between each station the tracks are divided into sections, each equipped with sensors reporting the passing of the train. In a nutshell, each train has to pass through several checkpoints between stations. The first objective of our distributed monitor is to check that the trains do obey some timing properties, the violation of them leading to possible catastrophic failure. The distributed monitors are located on computers (PC) at each station.

In this scenario, signals for abnormal behaviors are classified in two different classes: a *warning signal* corresponds to a violation of the specification that may not lead to a catastrophic failure and can be recovered, while a *error signal* corresponds to the system entering a state where a drastic action has to be taken to ensure safety of passengers.

On the monitoring side, the detection of a timing fault can be *i*) a warning (timing overhead on a single given section) that is part of the specification and does not require the monitoring system to raise an alarm, or *ii*) a real error signal that may lead to a catastrophic failure.

On the system operational side, a warning is handled by triggering a degraded mode of operation of the system (e.g. alternate route, skip of intermediate stations, etc.). The error leads the system to start a recovery action that will put it in a safety state, e.g., stopping the train on an alternate track.

Errors are raised by the monitor when detected. Additionally, if a warning was detected and the train does not use the degraded mode of operation, then an error must also be raised. However, the train may use the degraded mode even if neither warning nor error was detected.

B. Timing constraints

The monitoring system must signal an *error* if the train stays more than D units of time in a section, and a *warning* if it stays more than d time units in the first section; in this case, the degraded mode corresponds to a shorter path to try to recover with the time table.

As our monitoring tools deals with missing events, if the property "the train stayed more than d time units in the first section" does not hold, then an error will have to be raised if the degraded mode is not used.

C. Railway supervision system: Petri net modeling

Stations and sections are represented by places. The initial station is represented by p_1, the four sections are represented by p_2, p_3, p_4 and p_w and the final station by p_5. Sensors detecting the entrance in a section are represented by transitions $t_1, \ldots, t_4, t_w$. In other words, each time a sensor is activated, the corresponding transition will receive an event.

The nominal mode is represented in green and red in Figure 12(b). There are two paths: the long path p_1, p_2, p_3, p_4, p_5 is the nominal one, in green, while the short one p_1, p_2, p_w, p_5 is the degraded one, in red. The label $[0, D]$ between places and transitions indicates that a train cannot stay more than D units of time in any of the sections; if such a constraint were to be violated, then an error would have to be raised by the monitor.

D. Dealing with warning states

The last part of the Petri net, represented in blue in Figure 12(b), takes into account warnings. This blue part must be interpreted with a slightly different semantics: the firing of transitions must not be triggered by the reception of an event, but by the monitoring system as soon as they are fireable. We call these transitions with different semantics *logical transitions*, to distinguish them from event transitions. Notice that such transitions represent classical Petri nets transitions, i.e., they are not event-triggered, and thus their firing is straightforward and not described in this paper.

To detect if the long path can be taken by the train, the monitoring system needs to receive both events corresponding to t_1 and t_2. Indeed, if one of these two

events is missing, the monitor will ensure the train uses the short path, otherwise the monitoring system will raise an error.

We briefly show below how logical transitions allow to increase the expressiveness of the approach.

Let us assume that t_1 and t_2 were fired at time τ_1 and τ_2 respectively (with $\tau_2 > \tau_1$). As soon as both events are received by the monitor, one of the two transitions, r_1 or r_2 is fireable. Notice that the token in place q_1 has waited $\tau_2 - \tau_1$ time units since its creation at the firing of t_1. Thus, either r_1 or r_2 can be fired, but not both due to the disjoint timing constraints on arcs. Since none of these transitions are associated to an event, the monitor will fire the correct transition at time τ_2 (before τ_2, no token was present in place q_2).

To sum up, the monitoring system verifies that the system is in one of the two mutual excluding execution branches:

- if the train stayed less than d time units in the first section, then r_2 will be fired and tokens will appear in place q_3 and q_4, allowing the train to use its nominal path,
- if the train stayed more than d time units in the first section, transition r_1 is fired and blocks any firing of t_3 or t_4: the train is forced to run in the degraded mode (or fast path).

VI. Conclusion

MINOTOR is a contribution towards the on-line monitoring of distributed systems. MINOTOR improves the detection coverage of timing faults thanks to a model of the correct behaviors of the system that is animated at runtime through system events.

We impose the specification to be described as an timed-arc Petri net, a powerful formalism to express both distributed and timed behaviors.

Differently of static analysis, MINOTOR receives system events on the fly, and possibly out-of-order. To cope with this problem, we introduce the notion of *signed token*, i.e. we execute every transition associated to an event as soon as this event is caught by the monitoring system, no matter the current state of the Petri net. The actual monitoring of timing constraints and events ordering is performed a posteriori by checking respective dates of signed tokens.

The decoupling of transitions firing and timing constraints monitoring allows us to completely distribute the monitoring, as we show in the article. Our strategy is to associate to each transition, and each place in the Petri net a conceptual *thread*, in charge of executing an atomic sub part of the Petri net and verifying locally that timing constraints are valid.

In our experiments, complexity is manageable, in terms of memory footprint, but also in terms of execution time.

The definition of the model remains a complex issue. In some cases, such event-based specification may already be available to system developers, as it is one of the formalisms used for model checking, or static analysis of programs. This means that, Even for a complex physical system, the complexity of the model can be limited, abstracting all implementation details, while providing early detection mechanisms.

References

[BFR12] O. Baldellon, J-C. Fabre, and M. Roy. Distributed monitoring of temporal system properties using petri nets. In *SRDS*, 2012.

[BLS06] A. Bauer, M. Leucker, and C. Schallhart. Monitoring of real-time properties. *FSTTCS*, 2006.

[BR08] M. Boyer and O. H. Roux. On the compared expressiveness of arc, place and transition time Petri nets. *Fundamenta Informaticae*, 2008.

[CJ05] T. Chatain and C. Jard. Time supervision of concurrent systems using symbolic unfoldings of time petri nets. *Formal Modeling and Analysis of Timed Systems*, pages 196–210, 2005.

[FBHJ05] E. Fabre, A. Benveniste, S. Haar, and C. Jard. Distributed monitoring of concurrent and asynchronous systems. *Discrete Event Dynamic Systems*, 15(1):33–84, 2005.

[Hop91] R. Hopkins. Distributable nets. *Advances in Petri Nets 1991*, pages 161–187, 1991.

[JRR94] F. Jahanian, R. Rajkumar, and S.C.V. Raju. Runtime Monitoring of Timing Constraints in Distributed Real-Time Systems. *Real-Time Systems*, 7(3):247–273, 1994.

[MR10] P. Meredith and G. Roşu. Runtime Verification with the RV system. In *Proceedings of the First international conference on Runtime verification*, pages 136–152. Springer-Verlag, 2010.

[Ray10] M. Raynal. *Communication and Agreement Abstractions For Fault Tolerant Distributed Systems*. Morgan & Claypool, 2010.

[RFR08] T. Robert, J.C. Fabre, and M. Roy. On-line monitoring of real time applications for early error detection. In *PRDC*, pages 24–31. IEEE, 2008.

[RPC02] V. Ruiz, J. Pardo, and F. Cuartero. Translating tpal specifications into timed-arc petri nets. In *ICATPN'02*, pages 414–433. Springer-Verlag, 2002.

[Srb05] J. Srba. Timed-arc petri nets vs. networks of timed automata. In *ICATPN*, pages 385–402, 2005.

[ZSLL09] W. Zhou, O. Sokolsky, B. T. Loo, and I. Lee. DMaC: Distributed Monitoring and Checking. *Lecture Notes in Computer Science*, 5779:184, 2009.