Out-of-order Evaluation of Timed Petri Nets for Distributed Monitoring

Olivier Baldellon, Matthieu Roy, Jean-Charles Fabre

Out-of-order Evaluation of Timed Petri Nets for Distributed Monitoring

Olivier Baldellon ; Matthieu Roy ; Jean-Charles Fabre

LAAS ; CNRS, 7 avenue du colonel Roche, F-31077 Toulouse Cedex 4, France
Université de Toulouse ; UPS, INSA, INP, ISAE ; UT1, UTM, LAAS, F-31077 Toulouse Cedex 4, France
Email: {baldellon|roy|fabre}@laas.fr

***Abstract*—Modern embedded systems for safety critical applications, found in planes or cars, are based on real-time distributed networks and require high level of robustness, reliability and adaptability. In order to complement static off-line validation of such systems, this paper introduces a novel approach for on-line verification of behavioral properties expressed as a sub-family of timed Petri nets. We show that, for the subclass of *acyclic safe nets*, whose expressivity matches real-time systems constraints, it is possible to define a commutative and associative operation on states that enables out-of-order evaluation of the state of the Petri net, and thus a fully distributed evaluation of the behavior of the system under supervision. Additionally, when provided with a mapping between events and nodes of the distributed system, we show how to efficiently distribute the monitoring system on top of the applicative system, and provide algorithms for local evaluation and merging of information in order to detect as soon as possible when the system deviates from its specification.**

I. Introduction

The current and future evolution of embedded systems shifts them from the traditional small sized and centralized machines to widely *distributed* ones with timing constraints.

In the different means to provide dependability, *monitoring* systems and applications states seems a suitable and necessary tool to take into account at runtime many factors that could not be fully verified before execution. Indeed, the inherent complexity and unpredictability of distributed systems in interaction with their environment hardens the task of offline verification, due to components composition issues and high complexity.

Monitoring a system is the first step to be taken when programming dependable systems, since it is required to detect that something has failed in the execution, enabling possible reconfiguration of the system.

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 (distributed or not) in a centralized way has been an active research domain since few years now (see, e.g., [1], [2]), but these approaches cannot handle the case of systems that are both distributed and real-time. An interesting paper [3] proposed some mechanisms for distributed monitoring of simple properties, but this approach is not easily generalizable for more complex properties, as for example those with logical "or".

This paper introduces a new approach that allows to monitor in an efficient and fault tolerant way properties expressed with arc timed Petri nets. The main idea is to execute the Petri net on the fly, as soon as system events are received, by aggregating partial executions of Petri nets. We will show in the paper that this aggregation provides correct results independently of how (in which order) this aggregation is done. This *out-of-order* evaluation of Petri nets, which is the main contribution of our work, is an essential feature for enabling *distributed run-time evaluation* of properties expressed as Petri nets.

This paper is organized as follows. In Section II, we define the model of Petri nets used to express distributed and real time properties, and we express the link between system activity and the properties to be monitored. Then we present the semantics of our model, and define the failure model in Section III. Section IV presents an algebra that enables out-of-order execution of a Petri net, allowing speculative transition firings. Section V describes how to deploy the monitoring mechanisms on a distributed system and how to efficiently verify a property at runtime. Section VI compares our strategy with existing works.

II. System Model

A behavioral property on a system can be expressed with different formalisms, such as logics, (temporal) automata, Petri nets, etc. In this work, we focus on distributed and temporal properties, and we will suppose in the remainder of the paper that we are provided with a behavioral property expressed using a tailored version of arc-timed Petri nets.

First we define in Section II-A the Petri net model used for specifying distributed real-time properties. In Section II-B we refine the hypotheses made on the system and the model. The notion of timed event sequences that will be used to describe the observation done by monitors is explained in Section II-C. An example will be given and explained in Section II-D.

A. Acyclic Safe Nets

We will assume in this paper that the reader is familiar with Petri net formalism. Let us first recall some basic definitions on arc-timed Petri nets. In this paper the set B^A will be the set of functions from A to B.

Definition 1 — Arc-Timed Petri Net
An arc-timed Petri Net is 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 from the set of places to the set $\{0,1\}$, i.e., $M_0 \in \mathbb{M}_+$, where $\mathbb{M}_+ = \{0,1\}^P$ is the set of markings), ${}^\bullet(\cdot)$ and $(\cdot)^\bullet$ are the *backward* and *forward* incidence marking functions of $(\mathbb{M}_+)^T$, and I a function that associates to each arc between a place and a transition a time interval.

We say that there is an arc between a place p and an transition t if $p \in {}^\bullet t$. We will use in this paper the following notations: for every marking m and for every place p, we have "$p \in m$ iff $m(p) = 1$", ${}^\bullet p = \{t \mid p \in t^\bullet\}$, $p^\bullet = \{t \mid p \in {}^\bullet t\}$, and $^\circ(\cdot)$ is the transitive closure of ${}^\bullet(\cdot)$ on $P \cup T$.

An arc-timed Petri net is a classical Petri net where arcs between places and transitions are labelled with time intervals. Intuitively, a token can fire a transition if and only if the duration between the token's creation in the place and the firing of the transition belongs to the time interval that labels the arc between the place and the transition. For a more formal definition of the semantics, the reader can refer to [4] where different formalisms of timed Petri nets — including arc-timed Petri nets — are compared.

Definition 2 — Enabled and Fireable Transitions
Let M be a marking, a transition t is said to be *enabled* if $\forall p \in {}^\bullet t, M(p) = 1$. A transition t is said to be *fireable* if 1) t is enabled 2) for any $p \in {}^\bullet t$, the token in place p stayed in p during a time included in $I(p,t)$.

We will now introduce a new subclass of Petri nets, a generalization of occurrence nets that will be called *acyclic safe nets*. In the next definition, a 1-safe Petri net is a net whose all reachable markings contain at most one token per place; moreover an acyclic Petri net is a net such that $^\circ(\cdot)$ is irreflexive, in other words, for every place or transition e we have $e \notin {}^\circ e$.

Definition 3 — Acyclic Safe Nets
A arc-timed Petri net $\mathcal{N}$ is an acyclic safe net if:

- $\mathcal{N}$ is a 1-safe acyclic Petri net.
- $\forall p \in P, \quad |{}^\bullet p| \geq 1 \Rightarrow M_0(p) = 0$
- $\forall p \in P, \quad |{}^\bullet p| = 0 \Rightarrow M_0(p) = 1$
- $\forall t \in T$, there is a reachable marking that enables the transition t (*Quasi-liveness*).

A direct and important consequence of this definition is that for every correct execution and for every place p, at most one transition of ${}^\bullet p$ and at most one transition of $p^\bullet$ can be fired. This will be an interesting point for monitoring by making easy the association of the firing of the transition that created a token and the firing of the transition that consumed it. Indeed, with general Petri net the following scenario, that we want to avoid, is possible. Let p be a place, t_1 be a transition of ${}^\bullet p$ and t_2 be a transition of $p^\bullet$. If the two events "t_1 is fired at time 0" and "t_2 is fired at time 10" are received, we can not be sure that a token stayed in p for a duration of 10 (perhaps two firings was missed: t_2 at time 1 and t_1 at time 9). With acyclic safe nets, such ambiguous situations are not possible.

A drawback of this definition is the impossibility to represent looping processes, such as a periodic task. The special case of periodic tasks can be solved by launching the monitoring of the task periodically. Additionally, a loop with a bounded number of iteration can easily be unfolded. As this formalism is meant for modeling real time systems, unbounded loops that could result in a system hang cannot be represented in this model.

B. System, Model & Events

The principle of our approach is first to describe the system using an acyclic safe net $\mathcal{N}$ and then to execute this Petri net with a monitoring mechanism using a mapping that associates to each event a transition.

Transitions associated to events are called *event transitions* (or e-transition); the monitoring mechanism has to fire them when corresponding events are received. Transitions not associated to events are called *logical transitions* (or ℓ-transitions); the monitoring mechanism has to fire them as soon as they are fireable.

C. Timed Event Sequences

As explained in the previous subsection, our monitoring approach consists in: first associating to a sequence of observed events (e-transition) a sequence of events (that can correspond to e-transitions or ℓ-transitions), then associating to this sequence of events a Petri net execution. The first association will be described at the end of this paper in Section V. The second association will be described in the next section: Section III. The notion of sequence of event is introduced in this subsection.

Definition 4 — Timed Event
An timed event is a couple (t, τ) where t is a transition (not necessarily a e-transition) and τ a date (a real number).

A timed event can either be observed (an event associated to an e-transition occurred) or calculated (a ℓ-transition that has to be fired). The monitor has to execute a *sequence of timed events* $(e_i = (t_i, \tau_i))_{1 \leq i \leq n}$.

From now on we will assume that for any timed event e_i of a sequence $\mathcal{E}$, the two values t_i and τ_i are defined by the relation $(t_i, \tau_i) = e_i$.

Definition 5 — Correct Timed Event Sequences
Let $\mathcal{E} = e_1 \dots e_n$ a timed event sequence, we say that $\mathcal{E}$ is correct (or valid) if:

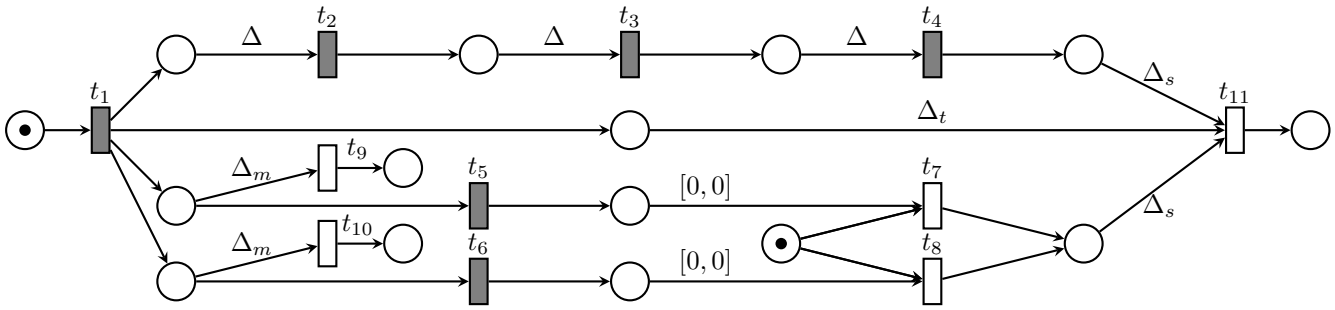

Fig. 1. An example of a distributed property: correctly displaying related information on an airplane control screen

- if $t_i \in {}^\circ t_j$ (t_j is a causal successor of t_i), then $i \leq j$.
- for every place p, if there are two timed events $e_i = (t_i, \tau_i)$ and $e_j = (t_j, \tau_j)$ such that $p \in {}^\bullet t_i$ and $p \in {}^\bullet t_j$ then $e_i = e_j$: a token can only be fired once.
- for every $e_i = (t_i, \tau_i)$, for all $p \in {}^\bullet t_i$,
  - if $|{}^\bullet p| = 0$, then $\tau_i \in I(p, t_i)$: p is a source.
  - if $|{}^\bullet p| \neq 0$ then there is a unique timed event $e_k = (t_k, \tau_k)$ such that $p \in t_k^\bullet$ and $\tau_i - \tau_k \in I(p, t_i)$: the timing of the event is correct.

Property 1 — Validity of Correct Sequences
Let $\mathcal{E} = e_1 \dots e_n$ be a correct timed event sequence. For every $j \in \{1, \dots, n\}$, if the transitions $t_1, \dots, t_{j-1}$ are fired at time $\tau_1, \dots \tau_{j-1}$ then the transition t_j is fireable at time τ_j.

Proof: Let us prove this result by induction. Let $\mathcal{E}$ be a correct timed event sequence.
Case 1: $\mathcal{E}$ has only one timed event $e = (t, \tau)$. Def. 5 implies that for all p in ${}^\bullet t$, $|{}^\bullet p| = 0$ (otherwise, there would be an event $e_k = (t_k, \tau_k)$ such that $p \in t_k^\bullet$ and thus we would have $t_k \in {}^\circ t$ and $e_k \neq e$: absurd !). Because no transitions were fired, the marking is the initial marking M_0, and hence , by Def. 3, $\forall p \in {}^\bullet t, M_0(p) = 1$. From this result we can know that t is enabled (Def. 2) and, as $\tau_k \in I(p, t)$, then we can conclude (Def. 2) that t is fireable.
Case 2: $\mathcal{E}$ contains more than one timed event. By induction we know that we were able to fire the transitions t_i at respective dates τ_i for $i \leq n-1$. Let M be the new marking obtained by firing of transitions $(t_i)_{i=1..n-1}$. Let p be a place of ${}^\bullet t_n$. By item 1 of Def. 5, no transition in $p^\bullet$ was fired. If $|{}^\bullet p| = 0$ then, due to the original marking M_0, $M(p) = 1$. If $|{}^\bullet p| = 1$, there is an event e_k such that the firing of t_k added a token in p and then $M(p) = 1$. Finally, we can conclude that t is enabled and, thanks to the second item of Def. 5, t is fireable. □

D. An Example of Distributed Property

Let us consider a critical display in a plane that provides information to the pilot. The screen is split in two parts: the left part (Primary Flight Display) and the right part (Navigation Display).

When the plane is in autopilot mode, the system periodically captures information on plane attitude (t_1). Based on this information, a regulation loop is executed, and the end of this processing sends an event t_2. Then, actuation orders of flight-related parameters are sent before event t_3, and, finally, the update of display information sends event t_4. On the left part, the Navigation Display, some navigation information like position information, active route and tracking are displayed by means of the Control and Display System.

Fig. 1 presents the related distributed property on the behavior of the overall display. Event transitions (e-transitions associated to system events) are colored in grey and ℓ-transitions (logical events or computations performed by the monitor) in white. There are six events in the system: $t_1 \dots t_6$.

Informally, this property can be explained as follows: the upper branch of the net (t_2, t_3, t_4) corresponds the left part of the screen and is a sequential process, while the lower branch (t_5, t_6, t_7, t_8) is the right part of the screen and presents a choice (two disjoint and exclusive possible paths). In the upper branch, sensing, processing, actuating and displaying have to be performed timely. The labels on arcs express the idea that each computation step has worst case execution time (WCET) in a time interval Δ. In parallel, in the lower branch corresponding to the Navigation Display, sensors capture information and, depending on some value ranges, information is delivered to the User Application if it belongs to the correct range, or triggers a recovery action when it is out of range. Notice that all branches are synchronized at the end to provide synchronous display of related events both on Primary Flight and Navigation Displays.

This is why the lower branch splits in two, the output of the display process depends on the occurrence pattern "value in range" (whether t_5 or t_6 happens). In practice the choice is described by the property: "t_5 or t_6 have to happen, but not both". As there is only one token in the place preceding t_7 and t_8, only one of those two transitions can be fired. Suppose that t_5, t_6 are fired in this order; then t_7 will be fired, and t_8 will not be enabled anymore, resulting in the death of the token between t_6 and t_8. The two places and the two transitions with the Δ_m time

interval forbid a token to stay forever before t_5 or t_6 if one of those two transitions is not fired. Indeed, a token that stays forever in a non final place is considered to be a dead token.

A third property that is expressed in this example is that the two branches must be synchronized (the display of the left and right part must be consistent): let τ_1 the date of t_4's execution and τ_2 the date of t_5 or t_6. If we want $|\tau_1 - \tau_2|$ to be less that a given value d_s, then, by choosing $\Delta_s = [0, d_s/2]$, a timing failure will occur whenever the two branches are not synchronized by at most d_s.

The middle branch with one place and one arc with interval Δ_t, expresses the property that the time that elapsed between the beginning (firing of t_1, or end of measurement) and the end (firing of t_9, or display of the screen) of the computation must be in Δ_t. Hence, setting $\Delta_m = [\sup(\Delta_t), \infty[$ frees up any blocked token and avoids dead tokens that do not correspond to any error at the end of the computation.

III. Semantics

In this section we describe how an execution of a Petri net is computed from a sequence of timed events. It is important to notice that we will not only consider correct event based sequences and that failures will be encountered. Instead of considering the execution itself, we will consider the final state of this execution. In this section, the state is computed in a *post-mortem* and centralized way. In Section IV we remove this assumption by providing a method to compute this state at runtime, in a distributed way.

A. Failures and States

We will assume two types of failures. The first one is *timing failures*. It corresponds to the case where a token in a place p has to fire a transition t while the age of this token is not included in the time interval $I(p,t)$. The second kind of failures, *missing events*, happen when a required event does not occur.

As the failure model includes missing events, when the timed event sequences will be executed on the Petri net, some transitions will have to be fired even if they are not enabled, i.e., tokens are missing in places before the transition. To handle this issue, we introduce in this section an extension of the set of marking by allowing one negative token in places where one token should be deleted. The new set of marking will be denoted $\mathbb{M}_\pm = \{-1, 0, 1\}^P$.

Definition 6 — State
A state is a tuple $(M, \epsilon_t, \epsilon_m, c_+, c_-)$ where M is a marking of $\mathbb{M}_\pm$, ϵ_t is the set of timing failure, ϵ_m is the set of missing events, c_+ is the positive calendar function that returns for every place p the date when a positive token appeared (a transition of ${}^\bullet p$ was fired) and c_- the negative calendar function that returns for every place the date when a negative token appeared (a transition of $p^\bullet$ was fired). The set of states is denoted $\mathbb{S}_\pm = \mathbb{M}_\pm \times \mathbb{E}_T \times \mathbb{E}_M \times \mathcal{C}^+ \times \mathcal{C}^-$.

The *timing failures* set $\mathbb{E}_T$ is composed of tuples (p, t, δ) defined by "a token stayed in a place p during δ and has to fire transition t while t is not fireable"

The *missing events* set $\mathbb{E}_M$ is composed of couples (place, sign). A couple $(p, +)$ is in $\mathbb{E}_M$ if a transition was not fired when it was supposed to be, letting a positive token that will never disappear in place p. A couple $(p, -)$ is in $\mathbb{E}_M$ if a not enabled transition t was fired, resulting in a negative token in place p.

The *positive calendar* function gives, for every place p, the date when the token appears, if any (the date when a transition of ${}^\bullet p$ was fired). The *negative calendar* function provides for every place a couple (date, transition): the date when the token left the place, and the transition it fired. Notice that these two functions are partial functions, since not every place is used in every execution.

B. Semantics

The semantics is a function that takes as an input a timed event sequence and returns a state (a marking, two sets of failures and a two calendars).

Let $\mathcal{E} = (e_i = (t_i, \tau_i))_{1 \le i \le n}$ a timed event sequence. We define the timing-failure function with $(p, t, \delta) \in failure(\mathcal{E})$ if and only if a token in place p fires a transition t after having waited δ while it is not allowed. In a more formal way, $(p, t, \delta) \in failure(\mathcal{E})$ iff $\exists i, j$ such that:

- $t = t_j$
- $p \in t_i^\bullet \cap {}^\bullet t_j$
- $\delta = \tau_j - \tau_i \notin I(p, t_j)$

Definition 7 — Semantics
Let M_0 be an initial marking and $\mathcal{E} = (e_i = (t_i, \tau_i))_{1 \le i \le n}$ a timed event sequence. The result of the execution $\mathcal{E}$ is a state $sem(\mathcal{E}) = (M, \epsilon_t, \epsilon_m, c_+, c_-)$ of $\mathbb{S}_\pm$ such that :

- M is the marking obtain from M_0 by firing transitions $t_1, \ldots, t_n$.
- ϵ_t is the set of timing failures: $\epsilon_t = failure(\mathcal{E})$
- ϵ_m is the set of missing events, defined by:

$$\epsilon_m = \big\{(p, +) \;\big|\; M(p) = 1 \wedge p^\bullet \neq \emptyset\big\} \cup \big\{(p, -) \;\big|\; M(p) = -1\big\}$$

- the calendar functions c_+ and c_- are defined as follows:

$$\forall i \in 1..n \qquad \forall p \in t_i^\bullet,\ c_+(p) = \tau_i \qquad \forall p \in {}^\bullet t_i,\ c_-(p) = (\tau_i, t_i)$$

IV. Algebra and Distributed Execution

We defined in Section III a semantics based on timed events. The computational result of such a semantics is a final marking and failure sets. We will see in this section how we can compute this result, not in a sequential way (as it is usually done) but in a fully distributed way. In other words, even if a transition t_1 has to be fired before t_2 in the Petri net, we provide a semantics that will allow to fire t_2 and t_1 in parallel.

The main idea is the following. Let S_0 be an initial state and $e_1, \dots e_n$ a timed event sequence. To compute the final state and the errors, we have to add e_1 to S_0 and then e_2 to the result and so on. Our approach consists in associating a state $[e_i]$ to each event e_i and introducing a new operation $\otimes$ to obtain a commutative group. Thus, to compute $S_0 \otimes [e_1] \otimes [e_2] \otimes [e_3] \otimes [e_4]$, a first node can compute $[e_2] \otimes [e_3]$ another one $[e_1] \otimes [e_4]$, the two of them can send the result to a last one that will compute $S_0 \otimes ([e_2] \otimes [e_3]) \otimes ([e_1] \otimes [e_4])$.

First we define a new set of markings that will allows us to get a commutative group. Then, to take into account time properties and potential errors, we extend this operation on states to obtain a commutative monoid structure.

A. Marking

Let $\mathcal{N}$ be an acyclic safe net and P the set of places of $\mathcal{N}$. We define the set of markings of $\mathcal{N}$ with $\mathbb{M} = \mathbb{Z}^P$. We introduced before two others definitions of the set of markings: $\mathbb{M}_+ = \{0,1\}^P$ and $\mathbb{M}_\pm = \{-1,0,1\}^P$. The previous definition was useful because in acyclic safe nets all possible markings are in $\mathbb{M}_\pm$. Before explaining why the new definition is interesting, let us generalize the classical $\oplus$ operation on markings in $\mathbb{M}_\mathbb{Z}$:

Definition 8 — Addition of marking
We define the operation $\oplus$ on $\mathbb{M}$ as follows: let M and M' be two markings. For any place p:

$$(M \oplus M')(p) = M(p) + M'(p)$$

Definition 9 — Marking of a transition
Let t be a transition. Its associated marking $\langle t \rangle$ is defined by, for every place p :

$$\langle t \rangle(p) = t^\bullet(p) - {}^\bullet t(p)$$

Intuitively, the idea behind the previous two formal definitions is the following: if, from a marking M, a fireable transition t is fired, then the new marking is the result of the operation $M \oplus \langle t \rangle$.

As an example, Fig. 2 shows five markings of the same Petri net: the initial marking, three transitions markings, and the final marking. It is easy to see that adding in any order the first four markings results in the final marking, obtained by the execution of t_1, t_2, t_3 from M_0. The two next properties formalize this idea.

Notice that, in general, adding two unrelated markings may result in meaningless marking (e.g., every marking of $\mathbb{M} \setminus \mathbb{M}_\pm$ is meaningless). Considering such meaningless computation is required only to provide a group structure.

Property 2 — Group structure
The couple $(\mathbb{M}, \oplus)$ forms an abelian (i.e. commutative) group.

Proof: The operation $\oplus$ is an internal operation on $\mathbb{M}$ (but was not in $\mathbb{M}_\pm$). The commutativity and associativity are direct consequences of the commutativity and associativity of the + operation in $\mathbb{Z}$. The neutral element is the empty marking defined by $\forall p \in P, 0_\oplus(p) = 0$. □

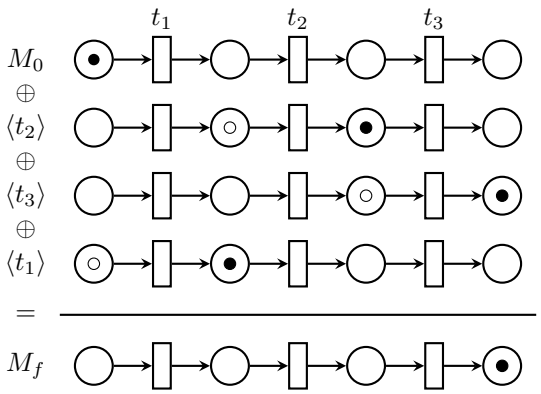

Fig. 2. Marking addition

Property 3 — Computation of the final marking
Let $\mathcal{N}$ be an occurrence net, M_0 a marking of $\mathcal{N}$ and $t_1, \dots, t_n$ a valid execution of $\mathcal{N}$ starting from M_0, then we have the following property:

$$M_0 \xrightarrow{t_1,\dots,t_n} M_0 \oplus \bigoplus_{i=1}^{n} \langle t_i \rangle$$

Proof: This is a direct consequence of the fact that firing a transition t from a marking M results in the addition of the two markings M and $\langle t \rangle$ (by definition of $\langle t \rangle$). □

B. Failure Detector $\diamond\mu$

The next step is to define an operation on states to obtain a commutative monoid. The main issue is that missing events cannot be determined using only algebraic properties: the detection of such failures depends on time. At runtime, to compute the state, the set of received events is not sufficient to compute the set of missing events (the exact set can only be computed post-mortem).

To circumvent this issue, we introduce in this section a failure detector $\diamond\mu$. Intuitively, the main goal of the oracle that compute $\diamond\mu$ is to determine the set of missing events ϵ_m that cannot be computed during the execution using only algebra on a subset of the timed events $e_1, \dots, e_n$ (in this example, $e_1, \dots, e_n$ are observed events, and the subset corresponds to partial information available at runtime).

Definition 10 — Failure detector $\diamond\mu$
Let $\mathcal{E}$ be an execution and $(M, \epsilon_t, \epsilon_m, c_+, c_-)$ its semantics. The oracle $\diamond\mu$ is defined by:

- at any time, $\diamond\mu$ is included in ϵ_m,
- after a known bounded delay, $\diamond\mu$ is equal to ϵ_m.

Notice that this failure detector definition is similar to the eventually perfect failure detector $\diamond P$ [5] with two differences: firstly, the definition holds for missed events instead of process crashes, and secondly, the delay from which the failure detector provides accurate results is bounded, with a known bound.

C. States

Information from markings does not include timing properties. In this section, we introduce a new notion of state by refining the definition of markings.

A state is a tuple of $\mathbb{S} = \mathbb{M} \times \mathbb{E}_T \times \mathbb{E}_M \times \mathcal{C}^+ \times \mathcal{C}^-$ where $\mathbb{M}$ is the set of markings defined in the previous section, $\mathbb{E}_T$ is the set of timing failures, $\mathbb{E}_M$ is the set of missing events and $\mathcal{C}^+, \mathcal{C}^-$ are the agenda function sets.

Let us define a new operation $\cup_f$ on calendar functions: $(c \cup_f c')(p) = \min(c(p), c'(p))$ (if $c(p)$ is not defined, we will note $c(p) = \perp$ and $\forall x, min(x, \perp) = x$). This function is an associative, commutative one.

Definition 11 — Addition of states
The operation $\otimes$ on $\mathbb{S}$ is defined by Algorithm 1.

Algorithm 1 The $\otimes$ operation

```
1:  function ⊗(S, S')
2:      (M, ε_t, ε_m, c_+, c_−) ← S
3:      (M', ε'_t, ε'_m, c'_+, c'_−) ← S'
4:      M'' ← M ⊗ M'
5:      ε''_t ← ε_t ∪ ε'_t
6:      ε''_m ← ε_m ∪ ε'_m
7:      c''_+ ← c_+ ∪_f c'_+
8:      c''_− ← c_− ∪_f c'_−
9:      for p ∈ P do
10:         if M(p) ≥ 1 and M'(p) ≤ −1 then
              ▷ addition of a positive and a negative token
11:           (d', t) ← c'_−(p)
12:           d ← c_+(p)
13:           if d' − d ∉ I(p, t) then        ▷ timing failure
14:               ε''_t ← ε''_t ∪ {(p, t, d' − d)}
15:           end if
16:       end if
17:       if M'(p) ≥ 1 and M(p) ≤ −1 then
              ▷ addition of a negative and a positive token
18:           (d, t) ← c_−(p)
19:           d' ← c'_+(p)
20:           if d − d' ∉ I(p, t) then        ▷ timing failure
21:               ε''_t ← ε''_t ∪ {(p, t, d − d')}
22:           end if
23:       end if
24:   end for
25:   return (M'', ε''_t, ε''_m, c''_+, c''_−)
26: end function
```

Definition 12 — Marking of a state
Let $e = (t, \tau)$ be an event. We associate to this event a state $[e]$ defined by $[e] = (\langle t \rangle, \emptyset, \emptyset, c_+, c_-)$ with c_+ and c_- defined by $\forall p \in t^\bullet, c_+(p) = \tau$ and $\forall p \in {}^\bullet t, c_-(p) = (\tau, t)$.

Definition 13 — Marking of a missing event
Let m be a missing event. We associate to this failure a state $[m] = (0_\otimes, \emptyset, \emptyset, \{m\}, \emptyset)$.

Theorem 1 — Computation of the final state
Let $\mathcal{N}$ be an acyclic safe net, S_0 the initial state of $\mathcal{N}$, $e_1, \ldots, e_n$ a timed event sequence, and ϵ_m the set of missing events of this sequence. Then:

$$sem(e_1, \ldots, e_n) = S_0 \otimes \bigotimes_{i=1}^{n} [e_i] \otimes \bigotimes_{m \in \epsilon_m} [m]$$

The previous Theorem uses ϵ_m, which cannot be computed instantaneously at runtime, but rather approximated by the failure detector $\diamond\mu$: after a known bounded delay, $\diamond\mu$ equals ϵ_m.

Proof: This result is a direct consequence of Definition 7 (semantics), Algorithm 1, and the definitions of $[e_i]$ and $[m]$. □

A set E_G is said to be generated by the $\otimes$ operation and the set E if: 1) $E \subset E_G$, 2) if e and e' are in E_G, then $e \otimes e'$ is in E_G. In other words, E_G is the closure of E with respect to $\otimes$.

Theorem 2 — Monoid Structure
Let $\mathcal{E} = e_1, \ldots, e_n$ be a timed event sequence and ϵ_m its set of missing events. Let $\mathbb{S}_\mathcal{E}$ be the subset of $\mathbb{S}$ generated by the $\otimes$ operation on the set $\{S_0\} \cup \{[e_i] \,|\, 1 \leq i \leq n\} \cup \{[m] \,|\, m \in \epsilon_m\}$.

$$(\mathbb{S}_\mathcal{E}, \otimes) \text{ is a commutative monoid.}$$

It is important to notice that the associativity property can be achieved only because we restrict ourself to acyclic safe nets: at most one token can appear in a place. Consider the case where two different tokens appear in a same place, one at the date τ_1 with the event e_1, and another one at the date τ_2 with the event e_2. Consider a third event e_3 that add a negative token to this place. The computation of $([e_1] \oplus [e_3]) \oplus [e_2]$ or the one of $[e_1] \oplus ([e_3] \oplus [e_2])$ will result in different states (in the first case, p contains one token that appeared in date τ_2, while in the second case, p contain one token that appeared in date τ_1).

Lemma 1 — Calendar function consistency
Let $\mathcal{E} = e_1, \ldots, e_n$ a timed event sequence and ϵ_m its set of missing events. Let $\mathbb{S}_\mathcal{E}$ be the subset of $\mathbb{S}$ generated by the $\otimes$ operation on the set $\{S_0\} \cup \{[e_i] \,|\, 1 \leq i \leq n\} \cup \{[m] \,|\, m \in \epsilon_m\}$. Let S_1 and S_2 be two states of $\mathbb{S}_\mathcal{E}$ with $S_i = (M_i, \epsilon_t^i, \epsilon_m^i, c_+^i, c_-^i)$; for any place p we have:
if $c_+^1(p)$ (resp. $c_-^1(p)$) and $c_+^2(p)$ (resp. $c_-^2(p)$) are both defined then $c_+^1(p) = c_+^2(p)$ (resp. $c_-^1(p) = c_-^2(p)$).

Proof of Lemma 1: This result is a direct consequence of the definition of a timed event sequence. As a positive token only appears at most once in a place, then there exists an unique i such that e_i created the positive token. By definition of $\mathbb{S}_\mathcal{E}$, if a state of this set has $c_+(p)$ defined, then the corresponding date is given by e_i. The reasoning is similar for negative token. □

Proof of Theorem 2: To prove this result, we have to prove that $(\mathbb{S}_\mathcal{E}, \otimes)$ is an internal, associative and commutative law with a neutral element. The fact that $\otimes$ is an internal law of $\mathbb{S}_\mathcal{E}$ is true by definition of $\mathbb{S}_\mathcal{E}$. The

commutativity property is a direct consequence of the symmetry of the code of Algorithm 1 (the two inputs S and S' have a symmetric same role). The neutral element is $(0_\otimes, \emptyset, \emptyset, \emptyset, \emptyset)$, and this can be easily checked with the algorithm. The only difficulty is to prove the associativity.

First of all, with the notation of Algorithm 1, the value of M'' (resp. c''_+, c''_- and ϵ''_m) depends only of those of M (resp. c_+, c_- and ϵ_m) and M' (resp. c'_+, c'_- and ϵ'_m). As the operations used to compute these values, $\otimes$, $\cup$ and $\cup_f$ are associative operations, we can deduce that for any states S_1, S_2 and S_3, the two values $S_1 \otimes (S_2 \otimes S_3)$ and $(S_1 \otimes S_2) \otimes S_3$ have the same marking, calendar functions and missing events set.

To conclude the proof, let $m = (p, t, \delta)$ a timing failure present in the timing failures set of $S_1 \otimes (S_2 \otimes S_3)$. We will demonstrate that m is present in the timing failures set of $(S_1 \otimes S_2) \otimes S_3$. In the following, c^i_- denotes the negative calendars function of S_i.

- *First case:* if m is in the failure set of S_1, S_2 or S_3, then m is trivially in $(S_1 \otimes S_2) \otimes S_3$.
- *Second case:* if m was computed during the operation $S_2 \otimes S_3$, it means that either $M_2(p) \geq 1$ and $M_3(p) \leq -1$ or $M_2(p) \leq 1$ and $M_3(p) \geq -1$ (lignes 10 and 17 of the algorithm). Without loss of generality we will only consider the first case: at least one positive token in p for the marking of S_2 and at least a negative one in p for the marking of S_3. There are still two subcases.
  - *First sub-cases*: $M_1(p) \geq 0$, then the marking of $S_1 \otimes S_2$ will associated more than one token in p and then m will be compute during $(S_1 \otimes S_2) \otimes S_3$.
  - *Second sub-cases*: $M_1(p) \leq -1$, then $c^1_-(p)$ is defined and thanks to Lemma 1 we know that $c^1_-(p) = c^3_-(p)$: thus, m will be compute during $S_1 \otimes S_2$.
- *Third case:* if m was computed during the operation $S_1 \otimes (S_2 \otimes S_3)$ then using a similar reasoning, we can also prove that m is a missing event of $(S_1 \otimes S_2) \otimes S_3$

We just proved that the timing failures set of $S_1 \otimes (S_2 \otimes S_3)$ is included in the one of $(S_1 \otimes S_2) \otimes S_3$. We can prove as well, using the commutativity property, that the timing failures set of $S_3 \otimes (S_2 \otimes S_1)$ is included in the one of $(S_3 \otimes S_2) \otimes S_1$. We have the equality of the timing failures sets of $S_1 \otimes (S_2 \otimes S_3)$ and $(S_1 \otimes S_2) \otimes S_3$. Hence we can conclude that $S_1 \otimes (S_2 \otimes S_3) = (S_1 \otimes S_2) \otimes S_3$. □

V. A Distributed Monitoring Protocol

Section III defined the semantics (i.e., the final state) of an observed timed event sequence. Section IV showed how this semantics can be computed in an unordered and distributed way (Theorem 1 and 2).

In this section we will show how we can use these two theorems to implement an efficient distributed monitor. We will describe in a first part the distributed model we consider and in a second part the distributed protocol.

A. System & Network Model

We assume we have a distributed system of monitors $\mathcal{M} = \{m_1, \ldots, m_n\}$. Some of the monitors can catch events, while some others are used for computation and routing only. Typically this system can be based on a pre-existing network that runs the distributed application we want to monitor. To simplify routing we will assume that a covering tree has already been computed and will be used for communication.

In our model, each event is caught by only one monitor. The event/monitor association is given by a function LOCATION from $\mathcal{M}^T$. The next figure gives an example of such an network. In the example, events associated to a monitor appear next to it.

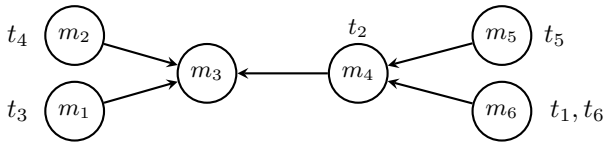

Fig. 3. The Monitor Network

This part will describe a protocol where events are caught by monitors that will forward information up to the root of the covering tree.

B. Deploying a Property

We saw in previous section that the computation of the final state can be done in an unordered and distributed way. We will show in this section how the computation can be mapped to the monitor tree. The main difficulty in our approach is not to compute the final state, but to find the timed event sequence. A part of the execution is known thanks to the monitor: events corresponding to e-transitions are caught by monitors, but the other part, i.e., events corresponding to ℓ-transitions, must be computed.

To compute such transition, we will assign to each ℓ-transition a monitor that will be in charge of firing this transition. We have already a location function that associates to each e-transition a monitor, we will now extend this function to associate to each ℓ-transition a monitor: this monitor will be then responsible of the firing of this logical transition.

We will present for that a distributed algorithm that starts from a monitor node and deploy from this node the set of transitions between the different nodes of the tree. This algorithm is a recursive algorithm: the root node shares the Petri net between itself and its children. Then the children start over with sharing with their own children the Petri net they received. The recursive algorithm end when leaves are attained. To explain the algorithms, let us first introduce a few new notions.

The notion of dependency is defined to allow us to answer the following question: "what is the minimal set of events $d(t)$ needed to know if a transition t can be fired?". To know if a transition t must be fired, we need to know

if there is a token in places just before this transition, i.e. in places $p \in {}^\bullet t$. Similarly, to know if there is a token in a place, we need to know if a token enters this place, i.e. a transition of ${}^\bullet p$ was fired, or if a token leaves this place, i.e. a transition of $p^\bullet$ was fired. In the following definition, $d(t)$ is the set defined by $d(t) = \{t' \in {}^\bullet p \cup p^\bullet \mid p \in {}^\bullet t\}$.

Definition 14 — Events Dependencies
Let t be a transition of a Petri net $\mathcal{N}$, we define the set of *events dependencies* of t in T as:

$$\text{dep}(t,T) = \begin{cases} \{t\} & \text{if } t \text{ is an } e\text{-transition} \\ \bigcup_{t' \in d(t)} \text{dep}(t', T \setminus \{t\}) & \text{otherwise} \end{cases}$$

From now on, we will denote $\text{dep}(t)$ the set $\text{dep}(t,T)$ where T is the complete set of transitions of the Petri net. For example $\text{dep}(t_8) = \{t_5, t_6\}$.

The previous definition describes the dependency of ℓ-transitions on e-transitions. The next definition shows how to extract a sub Petri net from another Petri net. The main idea is, as a node will only be responsible of the firing of some transitions, it only needs a subpart of the original net.

Definition 15 — Sub-Petri net
Let $\mathcal{N} = (P, T, {}^\bullet(\cdot), (\cdot)^\bullet, M_0, I)$ be an acyclic safe net and T' a subset of T. We define the sub-Petri net induce by T' by $\mathcal{N}_{|T'} = (P', T', {}^\bullet(\cdot), (\cdot)^\bullet, M_0, I)$ where $P' = \{p \in P \mid \exists t \in T' \text{ s.t. } p \in {}^\bullet t \cup t^\bullet\}$.

Based on dependancy definition (Definition 14) and sub-Petri net (Definition 15), Algorithm 2 computes an efficient partitioning of the Petri net.

The algorithm assumes to be executed on a given covering tree. On the current node, we denote $c_1, \ldots, c_n$ the trees whose root are children of the current node. For example, if the current node is m_3, then we will have $c_1 = \{m_1\}$, $c_2 = \{m_2\}$ and $c_3 = \{m_4, m_5, m_6\}$.

The partitioning procedure waits for the reception of an executable net and splits the transitions of this net in $n+1$ sets, with n the number of children. For this purpose, two arrays *events* and *logical* are computed such that the transitions of $events[i] \cup logical[i]$ will be fired by monitors of the subtree c_i if $i \neq 0$, $i = 0$ corresponding to transitions to be fired by the current node. The net that will be executed by the current node is stored in the variable *local_net*.

Let us explain this algorithm. It starts when a message START $\mathcal{N}$ is received, where $\mathcal{N}$ is the Petri net that the current node has to partition between itself and its children.

First (lines 5 to 11), it splits the set of event transitions in $n+1$ subsets, where n is the number of children. For each event transition, if this transition is caught by a monitor of the subtree c_i, then this transition is added to $events[i]$ (line 7). If there is no such child, it means that the events corresponding to the transition must be caught by the current node and thus, the transition is added to $events[0]$ (line 9).

Algorithm 2 Partitioning a Petri net

```
1:  procedure PARTITION
2:      wait reception of START 𝒩
3:      events ← [∅, ..., ∅]
4:      logical ← [∅, ..., ∅]
5:      for any t in T_e do
6:          if ∃i s.t. LOCATION(t) ∈ c_i then
7:              events[i] ← events[i] ∪ {t}
8:          else
9:              events[0] ← events[0] ∪ {t}
10:         end if
11:     end for
12:     for any t in T_ℓ do
13:         if ∃i s.t. dep(t) ⊆ events[i] then
14:             logical[i] ← logical[i] ∪ {t}
15:         else
16:             logical[0] ← logical[0] ∪ {t}
17:         end if
18:     end for
19:     local_net ← 𝒩_|event[0]∪logical[0]
20:     for i ∈ {1, ..., n} do
21:         send START 𝒩_|event[i]∪logical[i] to ROOT(c_i)
22:     end for
23: end procedure
```

Then (lines 12 to 18) it splits the set of logical transitions in $n+1$ sets as above. If an ℓ-transition can be computed in a child c_i (if every event transitions that are dependency of the ℓ-transitions are in $events[i]$), then this transition must be added to $logical[i]$. If no child can compute this transition, the the current node can and will compute it (line 16).

The last part of the algorithm consists in sending the Petri net that will be monitor by the child c_i to the root of c_i. The restriction of the Petri net to $\mathcal{N}_{|event[i] \cup logical[i]}$ allows us to be sure that, in the children's execution of the algorithm, if the line 9 or 16 is reached, then the children root will be responsible to the current transition.

C. The Monitoring Protocol

We describe here the runtime part of the monitoring protocol that performs its computation in a recursive way. The algorithm assumes that the monitoring system has a tree structure, and that logical transitions have been partitioned and assigned to monitors by Algorithm 2.

The main idea of the runtime evaluation is the following: each node computes a partial state s with the transition it is responsible of and the partial state computed by its children in the tree. When this partial state is computed, the current node sends this value to its father if it is not the root. If it is the root, then the computed state is the final state.

The $\diamond\mu$ failure detector, described in Definition 10, is an independent thread that sends messages MISSING(m) as soon as a missing event m is observed. Notice that the implementation of $\diamond\mu$ is omitted in this paper, but a simple implementation using timeouts can be used here: if a timeout expires too early, then the corresponding event will be ignored.

To compute its partial state in the monitoring protocol of Algorithm 3, each node initializes a variable s to $0_\otimes$ and updates it with the partial state received from its children (line 6 of Algorithm 3), the events and missing events it received (line 7 and 8), and updates the corresponding partial state. When all messages have been received (this property is indicated by the reception of a a STOP message, line 5), it sends its state to its father (lines 15 and 16) if the current node is not the root.

Every time the current state s is changed, the protocol searches if some logical transitions can be fired. The FIRABLE$_\ell$ function gives for every node the subset of logical transitions it is responsible of and that are fireable. As a token in a place $p \in {}^\bullet t$ cannot be fired before $c_+(p)+\min(I(p,t))$ (with min the function that returns the lower bound of an interval), the date when this transition is fired is the maximum of the different $c_+(p)+\min(I(p,t))$ values. In other words, a logical transition is fired as soon as it is fireable. Notice that this date is not computed using a local clock, but only using events dates.

We will assume that there is a timeout mechanism that can be triggered as soon as the first message, be it an event, a missing event or a state, is received. The main goal is to avoid waiting infinitely because of a failure on the network that will make impossible for a monitor node to receive a message that it is supposed to received. This timeout mechanism is hidden behind an oracle that sends a message STOP when all timeouts have been reached. In this case, the monitor will simply stop waiting for new messages.

Algorithm 3 The monitoring protocol

```
1:  procedure MONITORING
2:      s ← 0_⊗
3:      while waiting do
4:          wait reception of a message msg
5:          if msg = STOP then waiting ← FALSE
6:          if msg = STATE(σ) then s ← s ⊗ σ
7:          if msg = EVENT(e) then s ← s ⊗ [e]
8:          if msg = MISSING(m) then s ← s ⊗ [m]
9:          while ∃t ∈ FIREABLE_ℓ(s) do
10:             (M, ε_t, ε_m, c_+, c_−) ← s
11:             τ ← max{c_+(p) + min(I(p,t)) | p ∈ •t}
12:             s ← s ⊗ [(t, τ)]
13:         end while
14:     end while
15:     if root then return s
16:     else send STATE(s) to father
17: end procedure
```

D. Efficiency, Fault Tolerance & Robustness

A classical drawback of Petri nets is that events are defined in a sequential way: a transition can only be fired when all preceding transitions have been fired; if a single event is missing, it is not possible to execute the net. As our approach allows to fire a transition even if this transition is not enabled, then (1) every event transition will be fired, and (2) every ℓ-transition that can be fired will be fired, as soon as the e-transitions it depends on have been fired.

The protocol presented above is optimal in two ways. First, it is easy to see that the number of sent messages is equal to the number of edges in the monitor tree; if this tree is directly mapped to the communication infrastructure, then this solution is clearly optimal. Second, the final state is not computed at the root of the monitor tree, but the computation is distributed on the monitor network: a failure will be detected as soon as a node has enough information to compute it. This property is ensured by Algorithm 2, that assigns logical transitions to the first monitor that that receives the necessary information to fire such transitions (as defined by the events dependencies function of Definition 14).

Interestingly, this last property can be used to implement an efficient recovery process: as soon as a failure is detected, the treatment to manage this failure, e.g. a degraded mode of operation, can be done locally. In other words, the treatment will only imply monitors in the subtree whose root detected the failure.

With regards to quality of the detection mechanism, notice that the quality of the result is a growing function of the quality of both observation of the execution and monitor network. If all events that can be detected are detected and if the monitor network satisfies real time properties, then the final state computed will be the real final state of the system and will be computed in real time. On the other hand, if many events are not detected and if timeouts are chosen in a very pessimistic way, then the result will be computed more slowly and will be less accurate (i.e., detection of failures includes too many false positive ones).

Finally, one can notice that the underlying structure of our detection mechanism, a tree, is in no way a fault-tolerant data structure. To circumvent this issue, which is independent from the problem we are solving, traditional fault-tolerance mechanisms should be used to increase robustness of the tree. Two main strategies can be used: (1) a replication strategy applied to monitoring nodes of the tree (e.g., leader-follower replication or active replication), and (2) a classical node failure detection mechanism to monitor nodes healthiness and trigger a reconfiguration of the underlying tree in case of a failure.

VI. Related Works

As it was explained in this paper, our approach consists in transforming a Petri net execution in an algebraic computation that allows, thanks to the commutativity and associativity properties, to compute the execution in a distributed way, and thus, to implement a distributed monitor of real-time properties.

Some research works propose centralized monitoring of real-time properties [2] expressed using LTL formulae. Unfortunately, logics formulae are inherently centralized and cannot be easily adapted for distributed monitoring.

Some other works are compatible with the distribution of the monitoring as [6] or [3]; however, the way the distribution is done is different from ours. In our approach, a distributed property can be monitored in a distributed way while their strategy associates to each property a node of the network that will be in charge of this property: in other words, each property has a centralized monitor assigned. These approaches have a major drawback: the expressivity of the specification language used in [3] only allows conjunction of simple properties, while our model of Petri nets can express general logics formulae.

Interesting works on centralized monitoring of asynchronous [7] or real-time [8] distributed systems, are those of C. Jard *et al.*. There are three main differences with our work. First of all, the monitoring consists in finding from an events sequence a corresponding Petri net execution to "explain" the observation. In our formalism, executable nets, at most one transition is associated to each event and this problem is trivial, but those works consider Petri net where a simple event can correspond to many transitions. The second difference is that there are no ℓ-transitions: every transition is associated to an event.

The algebraic view of execution can be seen as a specialization and generalization for the case of acyclic safe nets of the algebraic view of Petri Nets defined in [9]. We first added the notion of negative tokens that enables commutativity and associativity of operations. We also generalized this algebraic structure for arc-timed Petri net by reasoning on states and not only on markings.

A work that must not be mistook with ours is the one on distributable nets [10]. This approach executes the Petri net in a distributed network and the execution should follow a *location function*: a transition can only be fired on a specific node. The resulting execution is still a sequential execution — if an event happens before an other, then the corresponding transitions must be fired in the same order — while our approach is by nature a concurrent evaluation, and hence easily distributable.

VII. Conclusion

This paper presented a new algebraic approach for out-of-order evaluation of timed Petri nets: the verification of the correctness of an execution is reduced to a simple computation in an associative and commutative set. This approach allows to implement distributed monitoring protocols in a simple way; thanks to the algebra, local computations are sufficient to compute the global state and to detect potential deviations from the specification.

Results presented in this paper represent a first important step towards the development of an efficient and realistic distributed and real-time monitoring tool. A major next step consists in generalizing our model to more expressive versions of arc timed Petri nets. There is also room for improvement in the definition of the state algebra. Currently, the final state contains in the two calendar functions the dates of all transitions firings. A future version of this algebra will simplify the two calendar functions: if two states are added and if the addition results in the firing of a transition, then the date of this firing could be removed from the resulting state, thus reducing states complexity.

Another important point is to precisely evaluate efficiency of the distributed verification algorithm. The one presented here is very efficient in terms of the number of exchanged messages, but can be slow with regards to the latency of detection. We are currently working on providing a true real-time and distributed protocol.

Last, the aim of our monitoring protocol is to detect failures; another line of research for future works is to make this protocol compatible with classical recovery mechanisms [11]. The main idea is to detect and recover failure locally, i.e. in the smallest possible subtree.

Acknowledgements

The authors want to thank Jean Fanchon for his insightful comments during the writing of this paper.

References

[1] T. Robert, J. Fabre, and M. Roy, "On-line monitoring of real time applications for early error detection," in *2008 14th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE, 2008, pp. 24–31.

[2] P. Meredith and G. Roşu, "Runtime verification with the rv system," in *Proceedings of the First international conference on Runtime verification*. Springer-Verlag, 2010, pp. 136–152.

[3] F. Jahanian, R. Rajkumar, and S. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.

[4] M. Boyer and O. H. Roux, "On the compared expressiveness of arc, place and transition time Petri nets," *Fundamenta Informaticae*, vol. 88, no. 3, pp. 225–249, 2008.

[5] T. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996.

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

[7] E. Fabre, A. Benveniste, S. Haar, and C. Jard, "Distributed monitoring of concurrent and asynchronous systems," *Discrete Event Dynamic Systems*, vol. 15, no. 1, pp. 33–84, 2005.

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

[9] J. Meseguer and U. Montanari, "Petri nets are monoids," *Information and computation*, vol. 88, no. 2, pp. 105–155, 1990.

[10] R. Hopkins, "Distributable nets," *Advances in Petri Nets 1991*, pp. 161–187, 1991.

[11] B. Randell and J. Xu, "The evolution of the recovery block concept," *Software Fault Tolerance*, pp. 1–22, 1994.