Distributed Monitoring of Temporal System Properties using Petri Nets
Olivier Baldellon, Jean-Charles Fabre, Matthieu Roy

To cite this version:

HAL Id: hal-01015494
https://hal.archives-ouvertes.fr/hal-01015494
Submitted on 26 Jun 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
**Distributed Monitoring of Temporal System Properties using Petri Nets**

Olivier Baldellon	extsuperscript{*}†, Jean-Charles Fabre	extsuperscript{*}†, Matthieu Roy	extsuperscript{*}†

* CNRS, LAAS, 7 avenue du colonel Roche, F-31440 Toulouse, France
† Univ de Toulouse, LAAS, F-31440 Toulouse, France

**Abstract**—Supervising a system in operation allows to detect a violation of system specification or temporal properties, and is the first step required by any reconfiguration mechanism.

In this work, we focus on run-time verification of temporal system properties in distributed and real-time systems. Based on a description of a property that includes events and temporal constraints, expressed as an arc timed Petri net, we automatically derive a monitoring system responsible for checking this property. The proposed approach enables the distributed verification of system properties.

Our contribution is twofold. On the theoretical side, we introduce a slight modification of the semantics of Petri nets to be able to execute it in partial executions and noisy observation environments. On the practical side, we show how to use this formal framework to provide a distributed and efficient monitoring system, and describe its current implementation.

**Keywords**—Distributed Monitoring; Online Verification; Petri nets.

**I. Our Approach**

**A. Petri Nets For System Properties**

An arc-timed Petri net is a tuple \((P, T, \cdot (\cdot), (\cdot), M_0, I)\), 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), \(\cdot (\cdot)\) and \((\cdot)\) are the backward and forward incidence functions that associate to each transition a set of places, 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 a transition \(t\) if \(p \in \cdot t\).

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

**B. Petri Net Execution**

Properties on system events that need to be monitored are expressed using the 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 \((t, \tau)\) where \(t\) represents the transition associated to an event and \(\tau\) 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, the sequence of events \((t_1, 10); (t_2, 15); (t_3, 21)\) in this order, it will first fire \(t_1\), storing that this firing was done at time 10. When the event corresponding to \(t_2\) is received, the transition \(t_2\) is fired because 15 − 10 ∈ [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 \(t_3\) six units of time \((\tau_3 - \tau_2 = 21 - 15 = 6)\), which is out of the specified interval \([0, 5]\). However, 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.
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 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.

*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 *“* \(t\) 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.

\[
\begin{array}{ccccccc}
p_0 & [0, \infty] & t_1 & p_1 & [3, 6] & t_2 & p_2 & [0, 5] & t_3 & p_3 \\
\end{array}
\]

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 \(p_3\). The removal of a 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.

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 all required tokens are present. In our approach, transitions are always fired speculatively, and the fireability property is checked *a posteriori.*

II. PROTOCOL & IMPLEMENTATION

A. The protocol

Due to lack of space, the formal description of the protocol is omitted. It implements a simulation of the Petri net by a set of thread, one for each place and one for each transition. The protocol is then a direct transcription of the execution semantics described in I-B with negative tokens. The protocol assigns a monitoring thread to each place and each transition of the Petri net. The communication graph of the Petri net of Figure 2 is presented in Figure 4: 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.

When a place thread receives a positive or negative token, it starts a timer. We make the assumption that the network is synchronous (bounded delay) and thus when a token is received, the place can compute the maximal delay to wait for reception of the negative version of this already received token.

\[(t_1, \tau_1) \quad (t_2, \tau_2) \quad (t_3, \tau_3)\]

Figure 4. Communications Graph

B. Implementation

We have developed a proof-of-concept of our approach in the Erlang language. The full code of our implementation, called Minotor, is available on the author website. The implementation is 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.

CONCLUSION

Differently of static analysis, a run-time verification system receives system events on the fly, and possibly out-of-order. To cope with this problem, we introduce the notion of virtual, or *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 verification 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 verification allows us to completely distribute the verification, 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 subpart of the Petri net and verifying locally that timing constraints are valid.

REFERENCES


