Event-Based Runtime Verification of Temporal Properties Using Time Basic Petri Nets

Matteo Camilli[1](✉), Angelo Gargantini[2], Patrizia Scandurra[2], and Carlo Bellettini[1]

[1] Department of Computer Science, Università degli Studi di Milano, Milan, Italy
{camilli,bellettini}@di.unimi.it

[2] Department of Management, Information and Production Engineering (DIGIP), Università degli Studi di Bergamo, Bergamo, Italy
{angelo.gargantini,patrizia.scandurra}@unibg.it

Abstract. We introduce a formal framework to provide an efficient event-based monitoring technique, and we describe its current implementation as the MahaRAJA software tool. The framework enables the quantitative runtime verification of temporal properties extracted from occurring events on Java programs. The monitor continuously evaluates the conformance of the concrete implementation with respect to its formal specification given in terms of Time Basic Petri nets, a particular timed extension of Petri nets. The system under test is instrumented by using simple Java annotations on methods to link the implementation to its formal model. This allows a separation between implementation and specification that can be used for other purposes such as formal verification, simulation, and model-based testing. The tool has been successfully used to monitor at runtime and test a number of benchmarking case-studies. Experiments show that our approach introduces bounded overhead and effectively reduces the involvement of the monitor at run time by using negligible auxiliary memory. A comparison with a number of state-of-the-art runtime verification tools is also presented.

Keywords: Runtime verification · Formal methods @ runtime · Timing analysis · Temporal properties · Petri nets

1 Introduction

Software systems are increasingly employed in most domains and activities, including safety critical ones. Therefore, the society increasingly relies on software, and unreliable or unpredictable behavior is becoming less and less tolerated. As a consequence, over the past years, the validation of software systems has become an increasingly important and active research area.

Event-based runtime verification [1] is the monitoring of running programs to verify the occurring events against the requirements. A particularly challenging aspect is the monitoring of temporal properties in the presence of strict time constraints. In fact, monitoring at runtime introduces overheads on the System Under Test (SUT) that may affect the correctness of the verified properties.

In this paper, we introduce a formal event-based runtime verification framework and we describe its current implementation as a Java software tool, so called MahaRAJA[1]. This framework enables the monitoring of Java programs, by evaluating the conformance of the concrete implementation with respect to its formal specification given in terms of Time Basic (TB) Petri nets [2] (or simply TB nets), a powerful temporal extension of Petri nets (PNs) for modeling concurrent/distributed systems with real-time constraints.

Although descriptive formalisms are very popular in runtime verification [3], the adoption of operational specifications, like in our approach, offers some advantages with respect to declarative specifications [4,5]. They are usually easier to write, visualize, understand, and allow for step-wise model refinement [6]. Moreover, although other operational formalisms such as timed-automata [7] or finite-state-machines [8] support the modeling of temporal or behavioral aspects, PNs-based approaches can be more concise and easier to use [9]. Furthermore, aspects such as messaging, communication protocols, which are commonly used in concurrent or distributed systems, can be difficult to model with the language primitives of timed-automata [10,11]. Finally, despite several state-based, logic-based, and event-based notations have been used for runtime verification [1], our work is the first attempt (to the best of our knowledge) exploiting the expressiveness of the TB nets for verifying temporal properties at runtime.

The MahaRAJA framework requires the SUT to be instrumented by using simple Java annotations on methods, in order to link the implementation to its formal model. Then, at runtime, the execution of the events of interest triggers the conformance verification of temporal properties. Rather than using heavy offline computation to predict the generation rate of possibly invalid events to estimate the maximum detection latency [12], we use an online approach that focuses on maintaining the analysis as lightweight as possible. MahaRAJA operates on and in conjunction with the SUT and it performs data collection and processing asynchronously with the SUT execution. The monitor and the SUT run concurrently on separated CPU cores using a buffer-based mechanism for communication. Our approach tries to bound the cost of executing the SUT instrumentation by having a bounded number of instructions executed upon the generation of possible invalid events. This runtime verification procedure is highly scalable because it does not depend on the size of the entire state space (often far larger than the model size [13]). It operates using just an occurring *event* and the *1-step reachability set* of the current model's state, thus using limited extra memory.

The tool has been applied to a number of benchmarking case studies [14] and we experimentally evaluated the runtime overhead, making it possible for a system designer to reason about the timing constraints of the SUT. The experiments show that MahaRAJA introduces limited monitoringoverhead and

[1] Monitoring at Runtime of temporAl properties on Java Applications.

limited detection latency, thereby opening up the possibility to adopt a *fast failing* approach or implement a *self-healing* procedure [15] in a latency-aware adaptation setting.

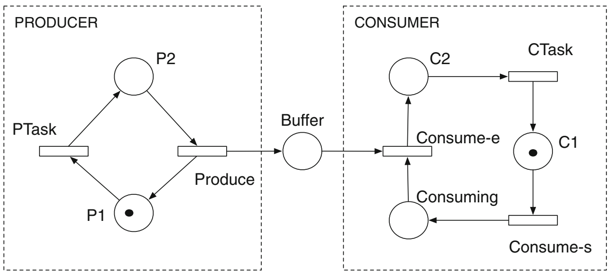
The paper is organized as follows. Section 2 introduces the proper background on TB nets. Section 3 introduces the formalization of our technique and Sect. 4 describes our current software implementation. Section 5 introduces our experimental evaluation of the runtime overhead, making it possible for a system designer to reason about the timing constraints of the SUT. Section 6 compares our monitoring framework with a number of state-of-the-art runtime verification tools, thus showing both advantages and disadvantages of our framework. Finally, Sect. 7 presents our conclusion and future directions of our work.

2 Background on Time Basic Nets

This section briefly introduces the TB nets formalism by means of a running example, i.e., the timed producer/consumer (P/C) model reported in Fig. 1.

TB nets are a formal model for distributed systems with real-time constraints. This modeling formalism is more expressive then other temporal extensions of PNs and it supports both time and functional extensions in a semantically clear and rigorous way [2]. Thus it represents an effective formal model to deal with specification of highly concurrent systems with real-time constraints.

The structure of a TB net [2,16] is a bipartite graph $N = (P, T, F)$, where P is the finite set of places (i.e., system state variables), T is the finite set of transitions (i.e., events causing state changes), $F \subseteq (P \times T) \cup (T \times P)$ is the flow relation. The pre/post-sets of $t \in T$ are ${}^\bullet t = \{p \in P : (p, t) \in F\}$ and $t^\bullet = \{p \in P : (t, p) \in F\}$, respectively.



Initial marking: $m_0 : \ P1\{T_0\}, C1\{T_0\}, \ T_0 = 0$

Time functions:

PTask	$[P1 + 1000, P1 + 2500]$
Produce	$[P2, P2 + 1000]$
Consume-s	$[C1, C1 + 1000]$
Consume-e	$[max(Buffer,\ Consuming{+}1000),\ max(Buffer,\ Consuming{+}1000) + 5000]$
CTask	$[C2 + 1000, C2 + 2000]$

Fig. 1. Producer-consumer TB net model.

The P/C example describes two processes that asynchronously interact through the place `Buffer`. After producing (respectively, consuming), the two processes perform some local activity (i.e., `PTask` and `CTask`).

In TB nets, tokens are enriched by timestamps recording their creation time. Each place can contain a multiset (bag) of tokens[2]. A *marking* (i.e., a representation of the system state) is a mapping $m : P \rightarrow Bag(\mathbb{R}_{\geq 0})$, that associates time-stamps to tokens in places (e.g., m_0 in Fig. 1).

Time constraints are associated with transitions: two (linear or linearizable) functions associated to each transition t define the lower and the upper bounds ($[lb, ub]_t$) of the interval of real values representing its possible firing times (e.g., $[P1 + 1000, P1 + 2500]$ associated with `PTask`). Tokens produced by the atomic firing of a transition are time-stamped with the same value. The actions of removing and creating tokens are performed instantaneously.

A *binding* of t is a function $b_t : {}^\bullet t \rightarrow \mathbb{R}_{\geq 0}$ that represents a set of time-stamps possibly causing t to be fired. The numerical interval $f_{b_t} : [lb(b_t), ub(b_t)]$ holds the possible *firing times* for b_t and it is evaluated by replacing each occurrence of a place p (free variable) with $b_t(p)$.

For instance, consider the transition `Consume-e` and the following binding: $b_{\texttt{Consume-e}}$: $\{\texttt{Buffer} \rightarrow 2000, \texttt{Consuming} \rightarrow 2500\}$. According to the time function of `Consume-e`, the firing times range over $[3500, 8500]$.

Starting from the marking m, a binding b_t is *enabled* if and only if $f_{b_t} \neq \emptyset$. A *firing instance* of t is a pair (b_t, τ) composed of an enabled binding and a real value $\tau \in f_{b_t}$. The firing of t results in a new marking m':

$$\forall p \in P, m'(p) = m(p) - i_{b_t}(p) + o_t^\tau(p)$$

where $i_{b_t}(p)$ is $1 \cdot b_t(p)$ if $p \in {}^\bullet t$, the null bag otherwise, $o_t^\tau(p)$ is $1 \cdot \tau$ if $p \in t^\bullet$, the null bag otherwise, and $+, -$ operators are extended to bags. This is denoted $m \xrightarrow{(b_t,\tau)} m'$.

For instance, the binding $b_{\texttt{Consume-s}}$: $\{\texttt{C1} \rightarrow 0\}$ is enabled in the marking m_0. Thus, the firing instance $(b_{\texttt{Consume-s}}, 450)$ is valid. The firing process produces a new marking m_1. In particular, it withdraws the token from place `C1` and it puts a fresh new token, time-stamped with the value 450, into `Consuming`.

The interval f_{b_t} can be interpreted in two different ways. The *weak* semantics states that t *can* fire at any instant in f_{b_t}. The *strong* semantics instead states that t *must* fire at any instant in f_{b_t}, unless it is disabled by a conflicting transition fired before the upper bound of f_{b_t} (refer to [2] for the details). Our running example adopts a strong semantics.

The marking m_n is *reachable* from m_0 if and only if there exists a *path* σ (sequence of firing instances and markings) such that:

$$\sigma = m_0 \xrightarrow{(b_{t_0},\tau_0)} m_1 \xrightarrow{(b_{t_1},\tau_1)} m_2, \ldots, m_{n-1} \xrightarrow{(b_{t_{n-1}},\tau_{n-1})} m_n$$

The transitions associated with the enabled bindings in m are called *enabled transitions* and they are denoted by $enab(m)$.

[2] $b \in Bag(X)$ is a map $X \rightarrow \mathbb{N}$, formally expressed as a weighted sum of X elements.

3 Event-Based Runtime Verification

This section introduces the formalization of our event-based monitoring approach. In order to abstract the behavior of a running program $\mathcal{P}$, let us introduce the observable components of $\mathcal{P}$, so called *action methods*.

Definition 1 (Action method). *Given a program $\mathcal{P}$, an action method is a subroutine performing a specific task, such that its execution is observed at runtime.*

The *action methods* are the events of interest that we want to observe and verify with respect to the expected behavior provided in terms of a TB net formal specification. During the execution of $\mathcal{P}$, we extract temporal information from the action methods depending on their own *action time*.

Definition 2 (Action time). *Given a program $\mathcal{P}$ and a set of action methods A, the* action time *function Γ maps action methods in A to a non empty set of elements in $\mathcal{T} = \{\texttt{initial}, \texttt{final}\}$.*

Intuitively, the *action time* determines the *moment* (i.e., time instant) at which we want to observe the action methods. $\Gamma(a) = \{\texttt{initial}\}$, implies that a is observed at its own invocation time. The temporal information extracted from the execution of a is a timestamp representing the *initial* time. Similarly, if $\Gamma(a) = \{\texttt{final}\}$, a is observed at its own *final* time; if $\Gamma(a) = \{\texttt{initial}, \texttt{final}\}$, a is observed both at invocation and termination time.

Given the observable components and the *action time* function, we use the notion of *timed trace* to abstract the behavior of a running real-time system.

Definition 3 (Timed trace). *Given a program $\mathcal{P}$ and a set of* action methods *$A = \{a_0, a_1, \ldots, a_m\}$, a timed trace is a finite sequence of events $\pi = e_0, e_1, \ldots, e_n$, such that each event $e \in \pi$ is a triplet $\langle a, g, v\rangle$, where:*

- *$a \in A$ is the action method that triggers the event. We denote it with $\alpha(e)$.*
- *$g \in \Gamma(a)$ is the moment associated with the event. We denote it with $\gamma(e)$.*
- *$v \in \mathbb{R}_{>0}$ is the timestamp associated with the event. We denote it with $\rho(e)$.*

As an example, consider the code excerpts reported in Fig. 2. They represent a JAVA implementation of the `Producer` and the `Consumer`, respectively, in a simple producer-consumer program. The `Consumer` calls the `consume` method that retrieves and removes a `Data` object from the `Buffer`, waiting if necessary until an element becomes available. Then it performs some additional tasks using the new element through the `consumerTask` method. The `Producer` creates a new `Data` object through the `producerTask` method and then it pushes the element into the `Buffer`.

The set of action methods is defined as $A = \{$*produce*, *producerTask*, *consume*, *consumerTask*$\}$, while the action type function is Γ: $\{$*produce* $\rightarrow \{\texttt{final}\}$, *producerTask* $\rightarrow \{\texttt{final}\}$, *consumerTask* $\rightarrow \{\texttt{initial}\}$, *consume* $\rightarrow \{\texttt{initial}, \texttt{final}\}\}$.

The rationale behind the Γ function is explained by means of the following example. Both the *produce* (Fig. 2, line 8) and the *producerTask* (line 12) action methods maps to {`final`} action time. In fact, the two action methods affect the behavior of the program at the end of their own execution: the *producerTask* method creates a new data element that becomes available at the end of the method execution; the *produce* puts the new data element into the buffer data structure and then terminates itself. Therefore, we want to observe only the final time of these action methods. Instead, the *consumerTask* action method (Fig. 2, line 13) processes the new data by launching an external asynchronous task. In this case we are just interested in knowing whether the external task is called in due time. Therefore, the *consumerTask* maps to {`initial`} action time. Finally, the execution of the *consume* action method (Fig. 2, line 8) causes the program to wait until a new element becomes available, which is consumed and returned at the end of the method execution. Hence, for each (multiple) execution of the *consume* action method, we want to observe both the initial and the final time. In fact, we may want to check that the consumer does not wait for available data more than a specific time limit.

The execution of the producer-consumer program can generate, for instance, the timed trace π reported in (1).

$$\begin{aligned}\pi = \ & e_0 : \langle consume, \texttt{initial}, 450\rangle,\\ & e_1 : \langle producerTask, \texttt{final}, 1100\rangle,\\ & e_2 : \langle produce, \texttt{final}, 1205\rangle,\\ & e_3 : \langle consume, \texttt{final}, 1650\rangle,\\ & e_4 : \langle consumerTask, \texttt{initial}, 2886\rangle.\end{aligned} \tag{1}$$

It is worth noting that *consume* occurs twice in π. In fact, the Γ function maps the *consume* action method both to `initial` and `final`, thus its own execution generates two different events timestamped with the initial and the final time, respectively.

```java
1  public class Producer
2      implements Runnable {
3    @Override
4    public void run() {
5      Data data = producerTask();
6      produce(data);
7    }
8    private void produce(Data data){
9      Buffer.getInstance()
10         .produce(data);
11   }
12   private Data producerTask(){
13     Data data = computeNext();
14     normalize(data);
15     return data;
16   }
17 }
```

```java
1    public class Consumer
2        implements Runnable {
3    @Override
4    public void run() {
5      Data data = consume();
6      consumerTask(data);
7    }
8    private Data consume(){
9      Data data = Buffer.getInstance()
10         .consume();
11     return data;
12   }
13   private void consumerTask(Data data){
14     log(data);
15     load(data);
16   }
17 }
```

Fig. 2. Java implementation of the Producer and Consumer.

Another important observation is that the program can perform inside the action methods different nested methods calls not belonging to A, therefore these are not observed at runtime. This allows us to build timed traces with different levels of granularity.

The construction of a *timed trace* is formalized as follows.

Definition 4 (Timed trace construction). *Given a running program $\mathcal{P}$, the set of action methods A and the action time function Γ, the* timed trace *π is constructed from the execution of each $a \in A$ such that:*

$$\forall g \in \Gamma(a), \langle a, g, v \rangle \in \pi,$$

where v is the timestamp associated with the moment *g.*

The timed trace constructed following Definition 4 includes only the events of interest defined by the action methods and the action time function.

To formalize the *conformance relation* between a running program $\mathcal{P}$ and its formal specification, let us introduce first the notion of *action method mapping.*

Definition 5 (Action method mapping function). *Given a TB net structure (P, T, E) and the set of action methods A associated with the program $\mathcal{P}$, the action method mapping function Λ associates each element $a \in A$ and each moment $g \in \Gamma(a)$ to a transition $\Lambda(a, g) \in T$.*

We use this mechanism to bind action methods in the implementation to transitions in the model. This way, the conformance verification can be performed by checking that all the events of a timed trace correspond to feasible firing transitions in the formal specification. The formalization is reported below:

Definition 6 (Path Conformance). *Given a timed trace π and an execution path σ of a TB net model, there exists a conformance relation between π and σ iff. for each $e_i \in \pi$, there exists $m_i \in \sigma$ such that:*

(i) $\Lambda(\alpha(e_i), \gamma(e_i)) = t_i$ (i.e., e_i is mapped to transition t_i)
(ii) $\rho(e_i) \in f_{b_{t_i}}$ (i.e., the timestamp of e_i belongs to the firing times of t_i)

Definition 7 (Model Conformance). *Given a timed trace π and a TB net model N and the Λ function, there is a* conformance relation *between π and N iff. there exists a feasible execution path σ of N, such that π conforms to σ, according to Λ.*

For example consider the timed trace π introduced in (1) and the following definition of the mapping function Λ :

$$\begin{array}{llll}
\Lambda(\mathit{producerTask}, \texttt{final}) & = \texttt{PTask} & \Lambda(\mathit{produce}, \texttt{final}) & = \texttt{Produce} \\
\Lambda(\mathit{consume}, \texttt{initial}) & = \texttt{Consume-s} & \Lambda(\mathit{consume}, \texttt{final}) & = \texttt{Consume-e} \\
\Lambda(\mathit{consumerTask}, \texttt{initial}) & = \texttt{CTask} & &
\end{array}$$

In this case, there exists a conformance relation between π and the producer-consumer TB net reported in Fig. 1. In fact, from the initial marking m_0 the transition `Consume-s` is enabled with the following binding $b_{\texttt{Consume-s}}$: $\{\texttt{P1} \to 0\}$. The timestamp $\rho(e_0) = 450$ belongs to $f_{b_{\texttt{Consume-s}}}$: $[0, 1000]$, thus we observe a valid event, and we can compute the next marking m_1, reachable from m_0 by firing the `Consume-s` transition at time 450.

$$m_1 : \mathit{P1}\{T_0\}, \mathit{Consuming}\{T_1\}; T_0 = 0, T_1 = 450.$$

The transition `PTask` is enabled from m_1 by the binding $b_{\texttt{PTask}}$: $\{\texttt{P1} \to 0\}$. The timestamp 1100, associated with the second event e_1 belongs to $f_{b_{\texttt{PTask}}}$: $[1000, 2500]$, thus we observe a valid event, and we can compute the next marking m_2, reachable from m_1 by firing the `PTask` transition at time 1100.

$$m_2 : \mathit{P2}\{T_1\}, \mathit{Consuming}\{T_0\}; T_0 = 450, T_1 = 1100.$$

And so forth, until we process the last action method. The complete path σ, such that π conforms to σ is:

$$m_0 \xrightarrow{(b_{\texttt{Consume-s}},450)} m_1 \xrightarrow{(b_{\texttt{PTask}},1100)} m_2 \xrightarrow{(b_{\texttt{Produce}},1205)} m_3 \xrightarrow{(b_{\texttt{Consume-e}},1650)} m_4 \xrightarrow{(b_{\texttt{CTask}},2886)} m_5$$

4 The MahaRAJA Framework

We implemented the runtime verification technique presented in the previous section as a Java library[3]. The main component of the library is the *Monitor*, i.e., a system that observes and analyzes an executing SUT (Java program) in order to verify its correctness by comparing the observed behavior (i.e., ordered timed trace) with an expected behavior (i.e., feasible execution path) of the TB model given in input as a PNML file [18]. The model can be easily generated using a graphical user interface that allows the user to create and edit arbitrary complex TB net models through simple drag and drop gestures.

The input program is linked to the formal specification exploiting the mechanism of Java annotations to map *action methods* to corresponding *transitions* (i.e., the mapping function introduced in Definition 5). The *Monitor* is executed in a separated thread and is composed of the following modules: the *Observer*, the *Analyzer* and the *Executor*.

The *Observer* module makes use of AspectJ [19] to observe code execution and trigger the verification of the *conformance relation*, performed by the *Analyzer* component. The framework defines a set of annotations[4] used to define the Γ *action type* function and the Λ *action methods mapping* function. The following annotations were inserted into the producer-consumer program:

```
@AfterType(trF="Produce")                  @AroundType(trI="Consume-s",trF="Consume-e")
private void produce(Data data){...}       private Data consume(){...}
@AfterType(trF="PTask")                    @BeforeType(trI="CTask")
private Data producerTask(){...}           private void consumerTask(Data data){...}
```

[3] The source code, binaries, and some runnable examples can be found at [17].

[4] They are recorded in class files by the compiler and retained by the virtual machine at run time, so they can be read reflectively by the *Observer* component.

Algorithm 1. Conformance verification procedure

```
1: function VERIFY($m, e$)
2:     $conformance$ = False
3:     if $\Lambda(e) \in enab(m)$ then
4:         $\tau = \rho(e)$
5:         for all $\langle b_t, f_{b_t} \rangle \in enab(m)$ s.t. $t = \Lambda(e)$ do
6:             if $\tau \in f_{b_t}$ then
7:                 $m' = computeNext(m, t, \tau)$
8:                 $addNext(m')$
9:                 $conformance$ = True
10:            end if
11:        end for
12:    end if
13:    return $conformance$
14: end function
```

As an example, the @`AroundType` annotation maps the *consume* action method to {`initial`, `final`} action times, thus observable both before and after its own execution. For each invocation we observe two events: the first event is bound to the `trI` transition; the second event is bound to the `trF` transition.

The execution of the methods annotated by @`BeforeType`, @`AfterType` and @`AroundType` are handled by @`Before`, @`After` and @`Around` AspectJ advice types [19], respectively, to generate the proper `inital` and/or `final` observable events. The *Observer* module inspects the execution of the SUT by using the facilities of AspectJ and generate observable events into the *event queue* by injecting additional code upon the execution of the action methods.

The *Analyzer* module incrementally builds the timed trace π through the *verification procedure* reported in Algorithm 1. For each occurring event e, extracted from the *event queue*, the *Analyzer* launches the verification procedure, passing as argument the current marking $m \in \sigma$ and the current event e. Thus, it verifies that in the input model, the transition t, retrieved by applying the Λ function, is enabled from the current marking m (line 3) and the time $\rho(e)$ belongs to f_{b_t} (line 6). If this condition holds, the *Executor* component updates the trace σ (line 7) creating a new reachable marking with the proper timestamp $\rho(e)$.

It is worth noting that, given an event e and a reachable marking m, there can be multiple enabled bindings for the transition t (line 5). In this case, for each binding, we compute a new reachable marking m' and we put it into the *reachability set* (line 8) representing all the valid next steps of σ. During the construction of the σ path, for each event e it is fundamental to maintain the entire 1-step *reachability set* for the transition t (instead of a single reachable marking), in order to avoid false alarms (i.e., unreal inconsistencies between the code and its specification) during the conformance checking. The Verify function is executed for each marking in the *reachability set*. If there does not exist any marking in the *reachability set* such that the *verification procedure* is successful, the *Analyzer* does not verify the *conformance relation* between π and σ, thus a *conformance failure* exception is thrown. This exception contains useful information about the throwing action method, along with the timestamp associated to this event and the set of enabled bindings (i.e., the expected events). The *Analyzer* module do not need to store the full history of both π and σ, thus it requires limited extra memory. Moreover, the verification procedure is scalable with respect to the SUT size, in fact its own time complexity (i.e., $\mathcal{O}(|enab(m)|)$) does not depend on either the model size or the entire state space, but just on the number of enabled bindings in the current marking.

To alleviate possible burst of the monitoring overhead, our framework makes use of the Java Thread Affinity [20] library to separate the execution of the SUT and the *Monitor* into different isolated CPU cores, decreasing the latency caused by suspending and resuming important running tasks. Moreover, MahaRAJA let the user define a *tolerance* that should be set to the expected monitor invocation overhead. The *tolerance* allows two levels of risk to be defined: *warning* and *error* corresponding to a timing constraint violation respectively in- and outside the *tolerance* range. By default the *tolerance* is disabled, in fact, its definition involves the evaluation of the monitor invocation overhead, which is not an easy task and it strictly depends on the underlying hardware/software environment.

In order to help the user to increase the confidence about the correctness of the SUT, the MahaRAJA software tool can be used in conjunction with JUnit to generate different monitored test cases. This way, the user can integrate our runtime verification technique with *assertion*s on variables and on specific *goal condition*s, given in terms of *time constraint* (i.e., a logical predicate formed by linear inequalities involving timestamps) on the observed *timed trace* [17].

The next section introduces our experimental results that could also be used as a guide to evaluate the runtime overhead in order to reason about the timing constraints of the SUT.

5 Experimental Validation

We validated the MahaRAJA framework by collecting data at runtime and performing a testing activity on a number of real-time benchmarking examples [14] summarized in Table 1: a simple producer-consumer (P/C) application, a cruise-control (CC) system, an automated teller machine (ATM) software system, an elevator (EL) controller and a factory (FA) automation distributed system.

Table 1. Case studies.

Case study	$\|P\|$	$\|T\|$	SLOC	Tasks	Frequency
P/C	5	4	208	3	4,19
CC	11	16	1185	4	2,65
ATM	12	25	1409	3	1,57
EL	18	24	1231	5	1,12
FA	14	12	996	10	1,09

The model size is reported in terms of number of places ($|P|$) and number of transitions ($|T|$). The SLOC column reflects the source lines of code number in the corresponding Java SUT. The *tasks* column contains the number of parallel threads (or process in case of distributed computing) composing the SUT. The *frequency* column reports the average number of monitor invocations per second.

The monitoring process ran in parallel with the SUT in a machine equipped with a Intel Xeon E5-2630 at 2.30 GHz CPU, 32 GB of RAM, the Ubuntu 14.04.3 LTS (GNU/Linux 3.13.0-39-generic x86_64) operating system with a completely fair scheduler [21], and the Java HotSpot 1.8 64-Bit Server virtual machine using the Garbage-First (G1) collector tuned to avoid full runs[5]. Data about runtime overhead is reported in this section. They were extracted from program executions monitoring $\sim 10^6$ events. The runtime overhead has been assessed considering the following metrics.

- **Monitoring Overhead:** The monitoring overhead is caused by the AspectJ instrumentation (AJO) and the monitor invocation overhead (MIO). Table 2 reports the average values (in μs) of these two different components, for each running case study. The average AJO values, introduced by the invocation of AspectJ advices, strictly depend on the byte code generated from the annotated program by using the ajc compiler [19]. Generally, we observed a lower AJO within @`Before` advices (i.e., events with `initial` action time) and a higher AJO within @`Arounde` advices (i.e., events with both `initial` and `final` action time). The order of magnitude of the measured AJO values is approximately 10 µs (see Table 2).
  The MIO (i.e., the time required to enqueue an occurring event into the *event queue*) does not depend on the action time. In general, both the MIO and the AJO have the same order of magnitude, but the average MIO is 50% lower. Thus, the overhead introduced by AspectJ dominates the overall monitoring overhead. Although the distribution of the MIO values for different programs are very similar, a different monitor invocation frequency (e.g., the CC frequency is 47% lower than the P/C one) impacts on the average MIO. For instance, the average P/C MIO is lower then the average CC MIO (approximately 16% lower).
- **Jitter:** The *Jitter* represents the deviance between the monitoring overhead values. The results reported in Table 2 show that the order of magnitude of the AJO jitter and the MIO jitter is the same (approximately 10 µs). We found that the AJO jitter, for all the action method types, is approximately 43% lower than the MIO jitter. While the AJO jitter strictly depends on the behavior of AspectJ at runtime, the MIO jitter depends on the state of the *Monitor* during the execution of the action methods. In fact, a suspended *Monitor* causes a burst of the MIO due to the time required by resuming it, during the enqueuing of an acton method into an empty *event queue*.
- **Detection latency:** Bounding the detection latency (DL) makes it possible for the *Monitor* to quickly recognize a conformance failure, thus making the SUT able to promptly react to a degraded situation though a recovery procedure.

[5] Additional information about the configurations of MahaRAJA and the JVM is available at [17].

Table 2 reports the average DL in μs. Our experience indicates the following trend: the higher the frequency is, the lower the DL is. This behavior is caused by the overhead of resuming a suspended *Monitor* thread. In fact, a low frequency implies an empty *event queue* almost all the execution time long. In this case, it is very likely to observe the *Monitor* resumption upon an incoming event. Therefore, although different programs lead to similar DL distribution, lower monitor invocation frequency results in more scattered DL values. The results obtained from our experiments show that MahaRAJA reacts to a conformance failure with a DL of the order of 1 ms.

– **Memory Overhead:** The memory overhead is the space used by the Java virtual machine to run and maintain the *Monitor* component. Table 2 shows that MahaRAJA requires negligible auxiliary memory (few KBytes on average). Gathered data shows that this value is related to the monitor invocation frequency: the higher is the frequency, the higher is the memory overhead. In fact, a high frequency implies the accumulation of events into the *event queue*.

Table 2. Monitoring overhead experiments results.

Case study		P/C	CC	ATM	EL	FA
AJO (μs)	Before	43.8	48.5	45.0	44.5	50.1
	After	59.0	51.5	47.4	52.3	52.8
	Around	53.4	53.8	53.1	61.4	58.3
AJO Jitter (μs)	Before	28.2	30.9	36.6	37.1	33.0
	After	27.2	24.8	19.6	20.6	20.3
	Around	22.8	12.5	26.2	27.6	34.5
MIO (μs)		28.0	23.6	24.0	23.0	24.1
MIO Jitter (μs)		45.5	45.4	44.8	50.4	45.7
DL (μs)		874.7	1221.9	1243.9	1274.4	1335.6
Memory (KB)		10838	5302	3503	2083	1734

6 Related Work and Comparative Evaluation

This section mentions the main approaches in the field of event-based runtime verification, and reports also a qualitative comparative evaluation of these tools for the runtime verification of Java programs. A preliminary quantitative comparison is available at [17].

CoMA [22] is a formal specification-based software tool that can continuously monitor the behaviors of a target Java program and recognize undesirable behaviors in the implementation with respect to its formal specification given in terms of Abstract State Machines (ASMs). Java PathExplorer (JPaX) [23] is a system for monitoring the execution of Java programs. The system extracts an execution trace (as a sequence of events) from a running program and verifies that the trace satisfies certain (past and future) LTL properties. Monitored bytecode is instrumented (by using JTrek) and an observer can check during runtime that the properties are never violated. The JAVA Monitoring and Checking (MaC) architecture [24] supplies two different specification languages: the Primitive Event Definition Language (PEDL) and Meta Event Definition Language (MEDL) allowing for a separation between the definition of the primitive events of a system and the system properties. Instrumented programs send an event stream to the event recognizer to identify higher-level activities, which are in turn processed to find property violations. HAWK [25] is a programming-oriented extension of the rule-based EAGLE logic [26] that has been shown capable of defining and implementing a range of finite trace monitoring logics, including future and past time temporal logic, extended regular expressions, and state machines. It is implemented as a Java library able to perform monitoring through a state-by-state comparison, avoiding to store the entire input trace.

Larva [27] is an event-based runtime verification monitoring tool for temporal and contextual properties of Java programs. The technique implemented in Larva makes use of dynamic communicating automata with timers and events (DATE) to describe properties of systems.

Monitored-oriented programming (MOP) [3] allows the source code of the SUT to be annotated with formal property specifications that can be written in any supported formalism. The formal specifications are translated in the target programming language. Thus, the obtained monitoring code can be used either at runtime or offline by checking traces recorded by probes. In this case, the violation handling mechanism is itself part of the design of the SUT, rather than an additional component on top of the system.

The analysis technique in [12] tries to estimate the rate of possible invalid occurring events and the maximum detection latency to realize predictable monitoring schemas. However, this is not always applicable due to different patterns in the occurrence of monitored events for different execution scenarios of the SUT [28]. An alternative approach used to decrease the monitoring overhead is time-triggered monitoring [28,29] which makes use of periodic sampling of the SUT state and different strategies to reduce the monitoring overhead by dynamically adjusting the sampling period.

Table 3 reports a comparative evaluation between MAHARAJA and some representative state-of-the-art runtime verification software tools. The following key features have been taken into account (for a more general comparison see [1]):

– *formalism*: it represents the formalism used to specify the SUT;
– *operational/descriptive*: it represents whether the tool uses a operational or descriptive formalism;
– *state/event-based*: state-based monitoring approaches rely on a state-by-state comparison, where a state stores the relevant data about the SUT;
– *exceptions*: it represents whether or not the user can express properties which include exception handling;
– *real-time*: it refers to the ability to verify quantitative temporal properties;

Table 3. Features comparison of different Java runtime monitoring tools.

Tool	MahaRAJA	Coma	Larva	Java-MOP	Java-MaC	Hawk	JPaX
Formalism	TB nets	ASMs	DATEs	various[a]	PEDL, MEDL	LTL, PLTL	LTL, PLTL
O/D[b]	O	O	O	O, D	D	D	D
S/E[c]	E	S	E	E	E	E	E
Exceptions	✓	✗	✓	✗	✗	✗	✗
Real-time	✓	✗	✓	✗	✓	✗	✗
Variables	✗	✓	✓	✓	✓	✗	✗
Self-awareness	✓	✓	✓	✓	✓	✗	✓
Testing	✓	✗	✗	✓	✗	✗	✗

[a]Depending on the plug-in: Finite State Machines, Regular Expressions, Context Free Grammar, PLTL, LTL, String Rewriting Systems. [b]Operational/Descriptive formalism. [c]State/Event-based approach.

- *variables*: it refers to the ability of monitoring value changes of variables;
- *self-awareness*: it refers to the capability of the monitoring system to return feedback to the SUT upon failure;
- *testing*: it represents the possibility to use the facilities of the monitoring framework to write test cases.

The results of our comparative evaluation show for each selected monitoring tool, the explicit support for the considered features. As we can see, the MahaRAJA framework has some interesting features, not directly supported by other tools. For instance, it allows both the runtime verification and testing of quantitative temporal properties. MahaRAJA does not support the monitoring of variables (Coma, Larva, Java-MOP and Java-MaC have this feature).

7 Conclusion

We presented an event-based runtime verification approach and its supporting tool MahaRAJA to verify temporal properties on Java programs. The proposed framework adopts TB nets to represent the desired behavior of the SUT, including real-time requirements. The designer annotates the source code to link Java methods to transitions of the model. Then, MahaRAJA exploits AspectJ to observe code execution and trigger the conformance verification at runtime. The usefulness of the approach has been assessed by monitoring a number of real-time benchmarking case-studies to discover both modeling and implementation faults. MahaRAJA focuses on the monitoring of timed events and its main limitation is that it does not support the monitoring of variables, although they can be easily checked during testing activity using MahaRAJA in conjunction with JUnit. Nonetheless, we believe that our approach represents a viable technique for checking temporal properties of Java programs with respect to their formal specifications. Our experience shows that the monitoring overhead can be numerically evaluated and we found AspectJ as the major bottleneck. For this reason, we plan to replace AspectJ with other efficient bytecode transformation techniques [30]. The auxiliary memory used by the instrumentation is negligible and a preliminary quantitative comparison with other representative state-of-the-art runtime verification software tools individuates MahaRAJA as the less invasive [17]. The detection latency is also limited, thus allowing for a prompt recover after a failure.

The quantitative evaluation lead us to consider MahaRAJA as a viable light-weight pluggable tool to support the verification at runtime of real-time self-adaptive systems [15,31]. We will explore this last topic in our future work.

References

1. Delgado, N., Gates, A.Q., Roach, S.: A taxonomy and catalog of runtime software-fault monitoring tools. IEEE Trans. Softw. Eng. **30**(12), 859–872 (2004)
2. Ghezzi, C., Mandrioli, D., Morasca, S., Pezzè, M.: A unified high-level Petri net formalism for time-critical systems. IEEE Trans. Softw. Eng. **17**, 160–172 (1991)
3. Chen, F., D'Amorim, M., Roşu, G.: A formal monitoring-based framework for software development and analysis. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 357–372. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30482-1_31
4. Arcaini, P., Gargantini, A., Riccobene, E.: Combining model-based testing and runtime monitoring for program testing in the presence of nondeterminism. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 178–187, March 2013
5. Liang, H., Dong, J.S., Sun, J., Wong, W.E.: Software monitoring through formal specification animation. Innov. Syst. Softw. Eng. **5**(4), 231–241 (2009)
6. Felder, M., Gargantini, A., Morzenti, A.: A theory of implementation and refinement in timed Petri nets. Theoret. Comput. Sci. **202**(12), 127–161 (1998)
7. Bengtsson, J., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. Springer, Heidelberg (2004)
8. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1), 77–111 (2000)
9. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. Technical report, Cambridge, MA, USA (1974)
10. Iglesia, D.G.D.L., Weyns, D.: MAPE-K formal templates to rigorously design behaviors for self-adaptive systems. ACM Trans. Auton. Adapt. Syst. **10**(3), 15:1–15:31 (2015)
11. Lee, W.J., Cha, S.D., Kwon, Y.R.: Integration and analysis of use cases using modular Petri nets in requirements engineering. IEEE Trans. Softw. Eng. **24**(12), 1115–1130 (1998)
12. Zhu, H., Dwyer, M.B., Goddard, S.: Predictable runtime monitoring. In: Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems, ser. ECRTS 2009, pp. 173–183. IEEE Computer Society, Washington, DC (2011)
13. Valmari, A.: The state explosion problem. In: Reisig, W., Rozenberg, G. (eds.) ACPN 1996. LNCS, vol. 1491, pp. 429–528. Springer, Heidelberg (1998). doi:10.1007/3-540-65306-6_21
14. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
15. Camilli, M., Gargantini, A., Scandurra, P.: Specifying and verifying real-time self-adaptive systems. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 303–313, November 2015

16. Bellettini, C., Capra, L.: Reachability analysis of time basic Petri nets: a time coverage approach. In: Proceedings of the 13th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, ser. SYNASC 2011, pp. 110–117. IEEE Computer Society, Washington, DC (2011)
17. Maharaja framework. http://camilli.di.unimi.it/maharaja/. Accessed Dec 2016
18. Hillah, L.M., Kordon, F., Petrucci, L., Trèves, N.: PNML framework: an extendable reference implementation of the Petri net markup language. In: Lilius, J., Penczek, W. (eds.) PETRI NETS 2010. LNCS, vol. 6128, pp. 318–327. Springer, Heidelberg (2010). doi:10.1007/978-3-642-13675-7_20
19. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001). doi:10.1007/3-540-45337-7_18
20. Chronicle Software: Java Thread Affinity Library (2016). http://chronicle.software/products/thread-affinity/. Accessed Jan 2016
21. Li, T., Baumberger, D., Hahn, S.: Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. SIGPLAN Not. **44**(4), 65–74 (2009)
22. Arcaini, P., Gargantini, A., Riccobene, E.: CoMA: conformance monitoring of Java programs by abstract state machines. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 223–238. Springer, Heidelberg (2012). doi:10.1007/978-3-642-29860-8_17
23. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods Syst. Des. **24**(2), 189–215 (2004)
24. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-MaC: a run-time assurance approach for Java programs. Form. Methods Syst. Des. **24**(2), 129–155 (2004)
25. d'Amorim, M., Havelund, K.: Event-based runtime verification of Java programs. SIGSOFT Softw. Eng. Notes **30**(4), 1–7 (2005)
26. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24622-0_5
27. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03240-0_13
28. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Time-triggered runtime verification. Formal Methods Syst. Des. **43**(1), 29–60 (2013)
29. Navabpour, S., Bonakdarpour, B., Fischmeister, S.: Path-aware time-triggered runtime verification. In: Qadeer, S., Tasiran, S. (eds.) RV 2012. LNCS, vol. 7687, pp. 199–213. Springer, Heidelberg (2013). doi:10.1007/978-3-642-35632-2_21
30. Mastrangelo, L., Hauswirth, M.: JNIF: Java native instrumentation framework. In: Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, ser. PPPJ 2014, pp. 194–199. ACM, New York (2014)
31. de Lemos, R., Garlan, D., Ghezzi, C., Giese, H.: Software engineering for self-adaptive systems: assurances (Dagstuhl Seminar 13511). Dagstuhl Rep. **3**(12), 67–96 (2014). http://drops.dagstuhl.de/opus/volltexte/2014/4508