Verification of Scheduling Properties Based on Execution Traces

Valérie-Anne Nicolas, Mounir Lallali, Stéphane Rubini, Frank Singhoff
Lab-STICC UMR 6285, Université de Bretagne Occidentale, UBL, Av. Le Gorgeu, 29200 Brest,
France; email: {surname.name}@univ-brest.fr

Abstract

Despite the use of scheduling analysis when designing hard real-time systems, some erroneous temporal behaviors may still occur at runtime. Monitoring the execution of the system during runtime is a way to spot faulty behaviors. We focus on inline and embedded monitoring for the verification of general but essential temporal properties: scheduling properties.
This paper presents an approach for the temporal scheduling properties verification part of monitoring. The proposed algorithm has been evaluated on a benchmark, detecting missed deadlines, priority inversions, deadlocks and locked resources, in keeping with scheduling analysis and simulation results.

Keywords: *monitoring, trace analysis, scheduling property verification, real-time system.*

Real-time system correctness depends on its logical and temporal correctness [1]. In the context of hard real-time systems, the system temporal constraints are essential and have to be met. The real-time scheduling theory provides methods and tools to describe, simulate such systems, and to verify temporal properties during the design stage. Despite the large amount of work in design stage modeling and verification of hard real-time systems, enhancing the overall system quality, some erroneous temporal behaviors may still occur at runtime.

Monitoring the execution of the system is thus mandatory to guarantee its integrity during its whole execution [2]. Moreover, to deal with hard timing constraints, the overall monitoring tool should be embedded into the system, while still being as non-intrusive as possible, and sufficiently efficient to adapt the system behavior, when needed, in a restricted delay. A monitoring tool observes the monitored system and builds a trace that constitutes a model of the real execution of the system. There is a number of trace models, depending on the kind of trace events, and in general closely related to the monitor tool, the type of monitored application, the intended properties or behaviors to observe. A processing module deals with the trace to obtain supervision information, for example compliance with specific temporal behaviors. A decision module may take action in line with supervision information, like ending the system execution for the most critical cases.

This paper presents an instance of a processing module applying temporal scheduling properties verification on execution traces as illustrated on Figure 1. We situate within the framework of the Cheddar scheduling analysis project and its associated Cheddar toolset including a scheduling analysis tool, a simulation tool [3], and a simplified architecture description language (called Cheddar ADL [4]). One of the output files when applying the simulation tool is the simulation trace file. This trace is the sequence of time-stamped events generated during simulation. The hereafter proposed verification module is based on the same system and trace models as in the Cheddar tool and the monitor introduced in [2].

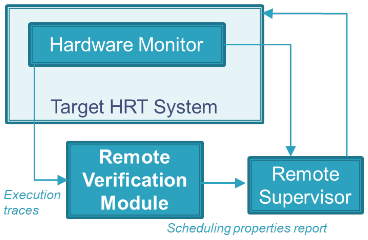

Figure 1: Verification module integration with the monitor

The paper is organized as follows: Cheddar system model, Cheddar trace model, and aimed temporal scheduling properties are described in Section 1. Next, we present the chosen approach to check temporal scheduling properties on execution traces in Section 2. In Section 3, the behavior of the proposed algorithm is illustrated on several simple examples. Then, related work is presented in Section 4. We finally conclude and point out upcoming improvements in Section 5.

1 System Model, Trace Model and Scheduling Properties

Figure 2 shows the verification module software architecture. The targeted systems for runtime monitoring are hard real-time systems on uniprocessor execution platform. The system model exported from the Cheddar ADL system model describes a system by a set of XML markup elements. Markup elements are dedicated to system hardware description (processors, cores, address spaces, scheduling parameters, etc.) and system software description (tasks, resources, resource sharing protocols, etc.) [4]. As an example, tasks are periodic and mostly characterized by their period, capacity, deadline, start time and priority. Resources are mainly characterized by their critical sections and the sharing protocol defining the access rules to the resource if it is shared by several tasks. The critical section for a resource *R* is the set of critical sections for the tasks sharing *R*. The critical section for a task *T*, using the shared resource *R*, is the time interval [*begin_time*, *end_time*] during which *T* uses *R*.

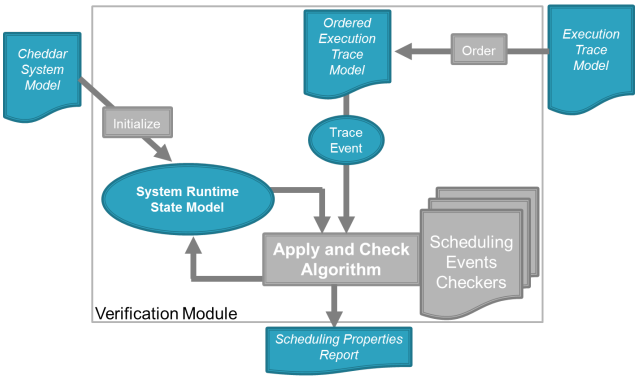

Figure 2: Software architecture of the verification module

The XML trace model produced by the Cheddar simulator or the monitor describes a system execution trace by a finite sequence of markup elements for time-stamped events. The types of events, numbering seven, come from the scheduling theory and describe the task states from the scheduling point of view. Events at *time i* for a task *T* (and resource *R*) are:

Task_Activation(i,T)	event sent out each time *i* where a task *T* is activated (ready to run)
Start_of_Task_Capacity(i,T)	event when *T* actually starts running at time *i*
Running_Task(i,T, T current_priority)	event when *T* runs at time *i* (with its priority that may change due to dynamic scheduling or resource sharing protocols)
Allocate_Resource(i,T,R)	event when a resource *R* is allocated to task *T* at time *i*
Wait_for_Resource(i,T,R)	event when a task *T* asks for an already used resource *R* at time *i*
Release_Resource(i,T,R)	event when a resource *R* is released by task *T* at time *i*
End_of_Task_Capacity(i,T)	event when a task *T* finishes its execution at time *i*

An extract of an XML execution trace model is presented in Figure 3 (in Section 2).

From the verification perspective, we are interested in scheduling properties of execution traces, numbering eight. For any given trace *Exe*, we focus on: *P_priority_inversion(Exe)*, *P_deadlock(Exe)*, *P_activation(Exe)*, *P_capacity(Exe)*, *P_deadline(Exe)*, *P_allocate(Exe)*, *P_unlock(Exe)* and *P_wait(Exe)*.
The properties *P_deadlock* and *P_priority_inversion* characterize the absence of the corresponding scheduling theory usual concepts.
In the simplest case and with a preemptive fixed priority scheduler, two tasks *T1* and *T2* are in deadlock if *T1* locks a resource *R1*, *T2* locks a resource *R2*, and *T1* waits for *R2* while *T2* waits for *R1*. Both tasks prevent each other from accessing the shared resources *R1* and *R2* and therefore are blocked, missing their deadlines.
Let see now an example of scheduling when a priority inversion occurs. A priority inversion occurs when two tasks *T1* (a low priority) and *T2* (a high priority) share a resource *R*, a third medium priority task *T3* uses no resource. *T1* begins and owns *R*, then *T2* is activated and preempts *T1*, *T2* later blocks waiting for *R* (still locked by *T1*). *T1* resumes its execution and *T3* is activated before *T1* has released *R*. *T1* is preempted by *T3*. At that point, *T3* (medium priority) can run and thus blocks *T2* (high priority), through *T1*, even though they share no resource.

We now define the other properties investigated in this paper.

P_activation(Exe)	holds true if for each system task, *Task_Activation* events occur at the accurate times (periodically from start time), with no missing or extra *Task_Activation* events in the whole trace *Exe*.
P_capacity(Exe)	holds true if each task job in the trace *Exe* runs exactly for the duration of its capacity.
P_deadline(Exe)	holds true if all task jobs in the trace *Exe* meet their deadlines.
P_allocate(Exe)	holds true if for each *Allocate_Resource(i,T,R)* event in the trace *Exe*, *R* is really needed by *T* at time *i*, *R* is free at time *i* and *i* is the required time for this event.
P_unlock(Exe)	holds true if for each system task in the trace *Exe*, owned resources are released at the required time, and in any case before deadline.
P_wait(Exe)	holds true if for each *Wait_for_Resource(i,T,R)* event in the trace *Exe*, *R* is really needed by *T* at time *i*, *R* is not free at time *i* and *i* is the event required time.

Brought together, all these properties give a fairly complete overview of the expected scheduling behavior of the system.

In the next Section we describe the algorithm for checking these properties, based on the system and trace models presented above.

2 Verification of Scheduling Properties on Execution Traces

The final objective of the verification module is to be embedded into the real-time system and run inline during the system execution. Its execution speed has thus to be compatible with that of the system. Another constraint, even if it is related, is that the monitored real-time systems may have non finite executions, or finite executions but with a great number of events. Therefore, during execution, the verification module does not take as input the whole trace, but a finite fixed size slice of it, using a transition buffer filled by the hardware part of the monitor. The direct induced impact is that the verification module execution time on one slice must be lower than the system execution time corresponding to the next trace slice, otherwise some trace events may be lost. For these reasons, the general frame of our verification algorithm is a one and only one pass through the trace.

As shown on the example of Figure 3 (which is a limited extract of events from a trace for conciseness), trace events are not fully ordered. This is especially the case for *Task_Activation* events. The *Task_Activation* event for a task *T* job is computed at the end of the previous task *T* job and immediately sent out stamped with the time of activation of the future task *T* job. An instance of that is the *Task_Activation* event at time *2* occurring in the trace before events stamped with time *0* or *1*. One may also note that several events may appear at the same time. It is quite common to find at the same time a *Task_Activation* event, a *Start_of_Task_Capacity* event and a first *Running_Task* event for the same task as illustrated by the example at time *0*. The events at a same time may also concern different tasks, as shown at time *3* with a *Wait_for_Resource* event for a first task and a *Release_Resource* event for a second task. There is a number of such possible combinations. Sorting the trace (according to time growing order) is thus imperative to allow to check the properties in a single pass through the trace. To order same time events, we define an order relation *event_order* on events, well suited to the kind of checked properties. For same time events, the order relation *event_order* states that:

$$\begin{aligned} &\mathit{End\_of\_Task\_Capacity} < \mathit{Task\_Activation} < \\ &\qquad\qquad \mathit{Start\_of\_Task\_Capacity} < \mathit{Running\_Task} \\ &\wedge\ \mathit{Running\_Task} < \mathit{Allocate\_Resource} \\ &\wedge\ \mathit{Allocate\_Resource} = \mathit{Wait\_Resource} = \mathit{Release\_Resource} \end{aligned}$$

meaning that, for example, a *End_of_Task_Capacity* event is considered as precedent a *Task_Activation* event even if they occur at the same time in the trace. As stated in Section 1, we target systems on uniprocessor execution platform and thus deal with one single execution trace on the processor. In that framework, the order relation *event_order* is compliant with the trace semantics. Actually, if a task job ends reaching its deadline (a *End_of_Task_Capacity(i,T)* event then follows the last *Running_Task(i-1,T,prio)* event), the task next job will be activated at the same time *i*, and possibly started and first runned also at the same time. On the contrary, by construction, the trace can not exhibit a *Task_Activation* (or *Start_of_Task_Capacity* or *Running_Task*) event and a *End_of_Task_Capacity* event at the same time for a same task job. Regarding resources, task resource allocation (or wait for resource) is first processed at the beginning of the first time unit where the resource is used by the task, whereas resource release is done at the end of the last using time unit. The same time resource related events can not be ordered in the absolute. Each pattern is specific, depending on the real use of resources by tasks. The order relation *event_order* states that the three resource related events are equal, which finally means that the order of these events in the initial trace is preserved.

```xml
<event_table>
 <mono_core_processor id="id_2">
  <scheduling_result>
   <result>
    <time_unit>0 </time_unit>
    <time_unit_event>
     <type_of_event>TASK_ACTIVATION</type_of_event>
     <activation_task ref="id_4"> </activation_task>
    </time_unit_event>
    <time_unit>2 </time_unit>
    <time_unit_event>
     <type_of_event>TASK_ACTIVATION</type_of_event>
     <activation_task ref="id_5"> </activation_task>
    </time_unit_event>
    <time_unit>0 </time_unit>
    <time_unit_event>
     <type_of_event>START_OF_TASK_CAPACITY</...>
     <start_task ref="id_4"> </start_task>
    </time_unit_event>
    <time_unit>0 </time_unit>
    <time_unit_event>
     <type_of_event>RUNNING\_TASK</type_of_event>
     <running_task ref="id_4"> </running_task>
     <current_priority>89</current_priority>
    </time_unit_event>
    <time_unit>1 </time_unit>
    <time_unit_event>
     <type_of_event>ALLOCATE_RESOURCE</...>
     <allocate_task ref="id_4"> </allocate_task>
     <allocate_resource ref="id_26"> </allocate_...>
    </time_unit_event>
    <time_unit>1 </time_unit>
    <time_unit_event>
     <type_of_event>RUNNING_TASK</type_of_event>
     <running_task ref="id_4"> </running_task>
     <current_priority>89</current_priority>
    </time_unit_event>
    <time_unit>2 </time_unit>
    <time_unit\_event>
     <type_of_event>START_OF_TASK_CAPACITY</...>
     <start_task ref="id_5"> </start_task>
    </time_unit_event>
    <time_unit>2 </time_unit>
    <time_unit_event>
     <type_of_event>RUNNING_TASK</type_of_event>
     <running_core>core1</running_core>
     <running_task ref="id_5"> </running_task>
     <current_priority>90</current_priority>
    </time_unit_event>
    <time_unit>3 </time_unit>
    <time_unit_event>
     <type_of_event>WAIT_FOR_RESOURCE</...>
     <wait_for_resource_task ref="id_5"> </...>
     <wait_for_resource ref="id_27"> </wait_...>
    </time_unit_event>
    <time_unit>3 </time_unit>
    <time_unit_event>
     <type_of_event>RELEASE_RESOURCE</...>
     <release_task ref="id_4"> </release_task>
     <release_resource ref="id_26"> </release_...>
    </time_unit_event>
    <time_unit>9 </time_unit>
    <time_unit_event>
     <type_of_event>END_OF_TASK_CAPACITY</...>
     <end_task ref="id_4"> </end_task>
    </time_unit_event>
   </result>
  </scheduling_result>
 <mono_core_processor id="id_2">
</event_table>
```

Figure 3: Extract of an XML execution trace model

We now describe the proposed algorithm for verifying scheduling properties in one pass from the time and *event_order* sorted trace. The different points where the properties are checked are depicted in the simplified outline of the algorithm presented in Figure 4. The algorithm is based on a representation of the system state at runtime (including task and resource states), and starting from an inactive initial state (built from the system model), simulates the execution represented by the trace, event by event. At the same time, and depending on the properties to verify, some checks are done on specific event occurrences and some others periodically at the end of each same time sequence of events. Periodic checks concern the tasks reaching the end of their period, and are needed to cope with possible missing events in the trace, such as missing *Task_Activation* events (thus contributing to *P_activation*) or *End_of_Task_Capacity* events. It also allows to complete the detection of undue locked resources (*P_unlock*), or task missed deadline detection (*P_deadline*). Otherwise, when dealing with a specific event, the algorithm checks that no property is violated by this event by calling the procedures associated to the event type

```
Algorithm: Apply&Check (system_runtime_state S, trace T)
foreach event E of trace T do
    state_update_with_event(S,E);
    switch E do
        case Task_Activation do
            P_activation_TActivEvt_Check(S,E);
            P_deadline_TActivEvt_Check(S,E);
        case Start_of_Task_Capacity do
            Start_of_Task_Capacity event error detection;
        case Running_Task do
            Running_Task event error detection;
            P_capacity_RunTaskEvt_Check(S,E);
            P_deadline_RunTaskEvt_Check(S,E);
            P_priority_inversion_RunTaskEvt_Check(S,E);
        case End_of_Task_Capacity do
            End_of_Task_Capacity event error detection;
            P_capacity_EndTaskCapaEvt_Check(S,E);
            P_deadline_EndTaskCapaEvt_Check(S,E);
            P_unlock_EndTaskCapaEvt_Check(S,E);
        case Allocate_Resource do
            P_allocate_AllocResEvt_Check(S,E);
        case Release_Resource do
            Release_Resource event error detection;
        case Wait_for_Resource do
            P_wait_WaitResEvt_Check(S,E);
            P_deadline_WaitResEvt_Check(S,E);
            P_deadlock_WaitResEvt_Check(S,E);
    end
end
Periodic_P_activation_Check(S);
Periodic_P_unlock_Check(S);
Periodic_P_deadline_Check(S);
End
```

Figure 4: Apply&Check Algorithm

adequate properties. These procedures are thus named *PropertyName_EventType_Check(S,E)*, meaning that they check that the event *E* of type *EventType* does not violate the property *PropertyName* in the system runtime state *S*. A specific type event has an impact on only some of the eight studied properties. Thus, only the procedures associated to the potentially impacted properties for the considered type of event are called. For example, a *Allocate_Resource* event may solely affect the *P_allocate* property whereas a *Task_Activation* event may affect the *P_activation* and *P_deadline* properties, and a *Wait_for_Resource* event the *P_wait*, *P_deadline* and *P_deadlock* properties. To give a more precise idea of the content of the checking procedures, here are some details about the *P_activation_TActivEvt_Check* and *P_deadline_TActivEvt_Check* procedures that are called when processing a *Task_Activation* event.

P_activation_TActivEvt_Check(S,E) :

- if the event *E* is a task first activation: checks that the event timestamp is not too late or too early,
- else checks that the previous task job activation event is not missing and that there is not extra activation event for the task in the interval.

P_deadline_TActivEvt_Check(S,E) :
checks that the previous task job did not miss its deadline.

The algorithm has been implemented in C in order to fit with the monitoring constraints: embedded into the system and efficiency.

In the next Section, the behavior of the algorithm is illustrated on several simple trace examples.

3 Evaluation of the Verification Module

The algorithm described in Section 2 has been evaluated on a benchmark of nine system and trace examples. This benchmark mainly comes from a Cheddar tutorial [5]. Each example is made of a system model and a trace model resulting from the Cheddar simulation tool. For all the examples, the verification algorithm results are compliant with Cheddar scheduling analysis and simulation tools. Among the nine examples, four exhibit erroneous behaviors (missed deadlines, deadlocks, priority inversions or locked resources).

For brevity, we here only present two mistaken examples whose system and trace models can be accessed online [6]. For each of them, we assume a preemptive fixed priority scheduling policy and priorities are assigned according to Rate Monotonic.
In the first example, a system with three periodic tasks, synchronous and with deadlines on request is considered.

Task	Period	Deadline	Capacity	Start time
T1	6	6	2	0
T2	8	8	2	0
T3	12	12	5	0

Tasks *T1* and *T3* share a resource S with mutual exclusion access: *T3* needs S during all its capacity, *T1* needs S during the 2nd unit of time of its capacity only. There is no specific priority inheritance protocol, blocked tasks are thus stored in a FIFO queue. The trace contains 75 events and expresses the system behavior over its feasability interval, that is from *0* to the *tasks periods Least Common Multiple (LCM)* as the tasks are synchronous [7], thus from time *0* to time *24*.
When executing our verification algorithm, a priority inversion between tasks *T1* and *T2* is detected at times *8* and *9*, and a missed deadline for the task *T1* is detected at times *12* and *13*.
Changing the sharing resource protocol by PIP (Priority Inheritance Protocol) leads to a correct behavior of the system, attested by the execution of the verification algorithm which finds no more errors.

The second example is a system with two asynchronous periodic tasks and one shared resource.

Task	Period	Deadline	Capacity	Start time
T1	20	20	10	0
T2	10	10	4	1

Tasks *T1* and *T2* share a resource *R1* with mutual exclusion access: *T1* needs *R1* from the the 1st unit of time of its capacity up to the 4th (included), and from the 3rd unit of time of its capacity up to the 6th (included). *T2* needs *R1* from the the 1st unit of time of its capacity up to the 2nd (included). There is no specific priority inheritance protocol. Here, tasks are not synchronous and the feasability interval is defined from *0* to the *maximum of tasks start times + 2 * LCM(tasks* *periods)* [7]. The trace contains 85 events and expresses the system behavior over its feasability interval, that is from time *0* to time *41*. When executing our verification algorithm, a deadlock on *R1* for task *T1* is detected at all times from *2*, a missed deadline for the task *T2* is detected at all times from *11* (while waiting for *R1*), an unlock error is detected on *R1* for *T1* at time *19* and *39*, a missed deadline for the task *T1* is detected at all times from *20* (while waiting for *R1*).

On this benchmark, results confirm that the whole set of considered properties give a fairly complete overview of the scheduling behavior of the system, similar to scheduling analysis and simulation results.

4 Related Work

Several works have been proposed for runtime verification/-monitoring of timed properties based on execution traces. [8] proposes a runtime verification framework for SoC (Systems on Chip) model. This framework allows the verification of temporal properties described in PSL (Property Specification Language), and the analysis of verification results. The authors of [9] present a software architecture based on Logic-Labeled Finite-State Machine (LLFSM) and regular expressions to perform runtime monitoring and verification of robotic system behaviors. [10] proposes a runtime verification approach for timed systems based on executable models. They define an on-the-fly conformance relation (between implementations and specifications) used for runtime verification, and they suggest an on-the-fly matching for timed traces. The proposed method has been implemented in an open-source toolkit which has been experimented on the verification of some units of different industrial microprocessors. [11] presents a predictive runtime verification framework for systems with timing requirements. Unlike the previous approaches, this predictive verification is related to a system which is not monitored as a black-box (some information about the system behavior is known).

Previous works propose their own verification framework and/or architecture that are not integrated as a part of the real-time system monitoring. In addition, these works deal with general temporal properties. In our case, we focus on scheduling properties verification for inline and embedded monitoring, and we aim at using our verification module as a part of an inline embedded health monitor.

5 Conclusion

In this paper, an approach for the verification of scheduling properties on uniprocessor hard real-time system execution traces has been presented. This verification module has been implemented in C and evaluated on a simple benchmark. Testing showed that verification module results were compliant with Cheddar scheduling analysis and simulation results, thus strengthening confidence in the algorithm pertinence and confirming that the set of considered properties gives an accurate overview of the expected scheduling behavior of the system. Currently, the verification module deals with one slice of execution trace. Next improvement is to enchain the processing of several execution trace slices. After what the objective is to use this verification module as a part of an inline embedded health monitor [2]. Further work is needed to evaluate the verification module on more consistent and realistic examples, so as to assess its efficiency when embedded into a real-time system.

Acknowledgments This work and Cheddar are supported by Brest Métropole, Ellidiss Technologies, CR de Bretagne, CG du Finistère and Campus France PESSOA programs number 27380SA and 37932TF.

References

[1] C. L. Liu and J. W. Layland, “Scheduling algorithms for multiprogramming in a hard-real-time environment,” *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[2] S. Rubini, V.-A. Nicolas, F. Singhoff, and J. Rufino, “A real-time system monitoring driven by scheduling analysis,” *RUME’18 Workshop*, June 2018.

[3] F. Singhoff, J. Legrand, L. Nana, and L. Marcé, “Cheddar: a flexible real-time scheduling framework,” *ACM SIGAda Ada Letters*, vol. 24, pp. 1–8, December 2004. ACM Press, New York, USA.

[4] C. Fotsing, F. Singhoff, A. Plantec, V. Gaudel, S. Rubini, S. Li, H. N. Tran, L. Lemarchand, P. Dissaux, and J. Legrand, “Cheddar architecture description language,” *Lab-STICC technical report*, 2014.

[5] F. Singhoff, “Tutorial about cheddar : an example of real-time scheduling analysis with cheddar,” *Lab-STICC technical report*, 2015.

[6] `http://beru.univ-brest.fr/svn/CHEDDAR/trunk/docs/publications/nicolas18`.

[7] J. Y.-T. Leung and M. Merrill, “A note on preemptive scheduling of periodic, real-time tasks,” *Information Processing Letters*, vol. 11, no. 3, pp. 115 – 118, 1980.

[8] L. Pierre and M. Chabot, “Assertion-based verification for soc models and identification of key events,” in *2017 Euromicro Conference on Digital System Design (DSD)*, pp. 54–61, Aug 2017.

[9] V. Estivill-Castro and R. Hexel, “Run-time verification of regularly expressed behavioral properties in robotic systems with logic-labeled finite state machines,” in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, pp. 281–288, Dec 2016.

[10] M. M. Chupilko and A. S. Kamkin, “Runtime verification based on executable models: On-the-fly matching of timed traces,” in *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, pp. 67–81, 2013.

[11] S. Pinisetty, T. Jron, S. Tripakis, Y. Falcone, H. Marchand, and V. Preoteasa, “Predictive runtime verification of timed properties,” *J. Syst. Softw.*, vol. 132, pp. 353–365, Oct. 2017.