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. 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. We situate within the framework of the Cheddar scheduling analysis project and its associated Cheddar toolset including a scheduling analysis tool, a simulation tool [2], and a simplified architecture description language (called Cheddar ADL [3]). 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.

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

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.) [3]. 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 $[\text{begin\_time}, \text{end\_time}]$ during which $T$ uses $R$.

The XML trace model produced by the Cheddar simulator describes a system execution trace by a finite sequence of markup elements for time-stamped events. The types of events come from the scheduling theory and describe the task states from the scheduling point of view. Events at time $t$ for a task $T$ (and resource $R$) are:
We now define the other properties investigated in this paper.\footnote{\textcopyright{} 2018 \textcopyright{} Ada User Journal. All rights reserved.}

\begin{tabular}{l}
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 \( i \) \\
\end{tabular}

An extract of an XML execution trace model is presented in Figure 1 (in Section 2).

From the verification perspective, we are interested in scheduling properties of execution traces. For any given trace \( \text{Exe} \), we focus on: \( P_{\text{priority inversion}}(\text{Exe}) \), \( P_{\text{deadlock}}(\text{Exe}) \), \( P_{\text{activation}}(\text{Exe}) \), \( P_{\text{capacity}}(\text{Exe}) \), \( P_{\text{deadline}}(\text{Exe}) \), \( P_{\text{allocate}}(\text{Exe}) \), \( P_{\text{unlock}}(\text{Exe}) \) and \( P_{\text{wait}}(\text{Exe}) \). The properties \( P_{\text{deadlock}} \) and \( P_{\text{priority inversion}} \) characterize the absence of the corresponding scheduling theory usual concepts. In the simplest case and with a preemptive fixed priority scheduler, two tasks \( T_1 \) and \( T_2 \) are in deadlock if \( T_1 \) locks a resource \( R_1 \), \( T_2 \) locks a resource \( R_2 \), and \( T_1 \) waits for \( R_2 \) while \( T_2 \) waits for \( R_1 \). Both tasks prevent each other from accessing the shared resources \( R_1 \) and \( R_2 \) 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 \( T_1 \) (a low priority) and \( T_2 \) (a high priority) share a resource \( R \), a third medium priority task \( T_3 \) uses no resource. \( T_1 \) begins and owns \( R \), then \( T_2 \) is activated and preempts \( T_1 \), \( T_2 \) later blocks waiting for \( R \) (still locked by \( T_1 \)). \( T_1 \) resumes its execution and \( T_3 \) is activated before \( T_1 \) has released \( R \). \( T_1 \) is preempted by \( T_3 \). At that point, \( T_3 \) (medium priority) can run and thus blocks \( T_2 \) (high priority), through \( T_1 \), even though they share no resource.

We now define the other properties investigated in this paper.

\begin{itemize}
\item \( P_{\text{activation}}(\text{Exe}) \) checks for each system task that \( \text{Task Activation} \) events occur at the accurate times (periodically from start time), with no missing or extra \( \text{Task Activation} \) events in the whole trace \( \text{Exe} \).
\item \( P_{\text{capacity}}(\text{Exe}) \) is true if each task job in the trace \( \text{Exe} \) runs exactly for the duration of its capacity.
\item \( P_{\text{deadline}}(\text{Exe}) \) states the absence of missed deadline for all task job in the trace \( \text{Exe} \).
\item \( P_{\text{allocate}}(\text{Exe}) \) checks for each \( \text{Allocate Resource} \) event in the trace \( \text{Exe} \) that \( R \) is really needed by \( T \). \( R \) is free and that this event occurs at the required time.
\item \( P_{\text{unlock}}(\text{Exe}) \) makes sure for each system task in the trace \( \text{Exe} \) that owned resources are released at the required time, and in any case before deadline.
\item \( P_{\text{wait}}(\text{Exe}) \) verifies for each \( \text{Wait for Resource} \) event in the trace \( \text{Exe} \) that \( R \) is really needed by \( T \). \( R \) is not free and that this event occurs at the required time.\end{itemize}

Brought together, all these properties give a fairly complete overview of the 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 below (which is a limited extract of events from a trace for conciseness), trace events are not fully ordered. This is especially the case for \( \text{Task Activation} \) events. \( \text{Task Activation} \) events for a task \( T \) job are 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 \( \text{Task Activation} \) 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 \( \text{Task Activation} \) event, a \( \text{Start of Task Capacity} \) event and a first \( \text{Running Task} \) 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 \( \text{Wait for Resource} \) event for a first task and a \( \text{Release Resource} \) event for a second task. There is a number of such possible combinations. Sorting the trace (according to time growing order) is thus imperative in order to check the properties in a single pass through the trace. To sort same time events, we define an order relation \( \text{event order} \) on events, well suited to the kind of checked properties. For same time events, the order relation \( \text{event order} \) states that:

\[
\text{End of Task Capacity} \prec \text{Task Activation} \prec \\
\text{Start of Task Capacity} \prec \text{Running Task}
\]

\[
\land \text{Allocate Resource} = \text{Wait Resource} = \text{Release Resource}
\]

The order relation \( \text{event order} \) is compliant with the trace semantics. Actually, if a task job ends reaching its deadline (a \( \text{End of Task Capacity} \) event then follows the last \( \text{Running Task} \) 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 \( \text{Task Activation} \) (or \( \text{Start of Task Capacity} \) or \( \text{Running Task} \) event and a \( \text{End of Task Capacity} \) 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.

We now describe the proposed algorithm for verifying scheduling properties in one pass from the time and event_order sorted trace. The algorithm is based on a representation of the system state (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 event occurrences or periodically at the end of each same time sequence. 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. It also allows to complete the detection of undue locked resources (P_unlock), or task missed deadline detection (P_deadline). The different points where the properties are checked are depicted in the following simplified outline of the algorithm (described in Figure 2). The algorithm has been implemented in C in order to fit with the monitoring constraints: embedded into the system and efficiency.

Algorithm: properties_checking (system S, trace T)

foreach event E from trace T do
  switch E do
    case Activation do
      | P_activation; P_deadline;
    case Start do
      | Start event error detection;
    case Running do
      | Running event error detection;
        P_capacity; P_deadline;
    case End do
      | End event error detection; P_capacity;
        P_deadline; P_unlock; P_priority_inversion;
    case Allocation do
      | P_allocate;
    case Release do
      | Release event error detection;
    case Wait do
      | P_wait; P_deadline; P_deadlock; P_unlock;
  end
end
periodic_check;

Figure 2: Properties Checking Algorithm

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 [4]. 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. 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.
Tasks $T_1$ and $T_3$ share a resource $S$ with mutual exclusion access: $T_3$ needs $S$ during all its capacity, $T_1$ 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 an hyper-period, that is from time 0 to time 24.

When executing our verification algorithm, a priority inversion between tasks $T_1$ and $T_2$ is detected at times 8 and 9, and a missed deadline for the task $T_1$ 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 periodic tasks and one shared resource.

<table>
<thead>
<tr>
<th>Task</th>
<th>Period</th>
<th>Start time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>$T_2$</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>$T_3$</td>
<td>12</td>
<td>5</td>
</tr>
</tbody>
</table>

Tasks $T_1$ and $T_2$ share a resource $R_1$ with mutual exclusion access: $T_1$ needs $R_1$ 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). $T_2$ needs $R_1$ from the the 1st unit of time of its capacity up to the 2nd (included). There is no specific priority inheritance protocol. This system hyper-period is 20 but we studied a longer trace of 85 events from time 0 to time 40.

When executing our verification algorithm, a deadlock on $R_1$ for task $T_1$ is detected at all times from 2, a missed deadline for the task $T_2$ is detected at all times from 11 (while waiting for $R_1$), an unlock error is detected on $R_1$ for $T_1$ at time 19 and 39, a missed deadline for the task $T_1$ is detected at all times from 20 (while waiting for $R_1$).

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 works

Several works have been proposed for runtime verification/monitoring of timed properties based on execution traces. [5] 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 [6] 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. [7] 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. [8] 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 confirming that the set of considered properties gives an accurate overview of the scheduling behavior of the system.

Currently, the verification module deals with one slice of execution trace. Next improvement is to chain the processing of several execution trace slices.

The objective of this project is to use this verification module as a part of an inline embedded health monitor. 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


