Computing a Language-Based Guarantee for Timing Properties of Cyber-Physical Systems

Neil Dhruva
Birla Institute of Technology and Science, Pilani, India

Pratyush Kumar, Georgia Giannopoulou, Lothar Thiele
Computer Engineering and Networks Laboratory, ETH Zurich

Abstract—Real-time systems are often guaranteed in terms of schedulability, which verifies whether or not all jobs meet their deadlines. However, such a guarantee can be insufficient in certain applications. In this paper, we propose a method to compute a language-based guarantee which provides a more detailed description of the deadline miss patterns of an observed task. The only requirement of our method is that the timing behavior of the real-time system be modeled by a network of timed automata. We compute the language-based guarantee by constructing an equivalent finite state automaton in an iterative manner, using a counter-example guided procedure. We illustrate the language-based guarantee for two applications: design of a networked control system and scheduling in a mixed criticality system. In both cases, we show that the language-based guarantee leads to a more efficient design than the schedulability guarantee.

I. INTRODUCTION

Cyber-Physical Systems (CPS) interconnect computational and physical sub-systems which are often designed and analyzed under different theoretical foundations. Consider the example of a networked control system [1]. The timing properties of the communication network may be analyzed with real-time schedulability tests which guarantee whether all jobs meet their deadlines. On the other hand, given the worst-case sensor-to-actuator delay, the physical plant can be stabilized with a controller design for time delayed feedback systems [2]. A key challenge in the composition of such sub-systems is the provision of tight and compact guarantees.

The real-time systems community has primarily focused on providing schedulability as a guarantee. From the perspective of the real-time system, if a task-set is schedulable, i.e., if all jobs are guaranteed to finish before their deadlines, then and only then, will all other sub-systems perform correctly. A large body of research has investigated methods and tools to verify such a guarantee under different assumptions of task models, scheduling policies and resource considerations.

However, schedulability can be insufficient in certain settings. For the networked control system example, the physical plant may be able to withstand a few missed deadlines. Indeed, in [3], the authors show that a well-defined class of deadline hit and miss patterns can guarantee stability of a physical plant. Hence, it is pertinent to look for richer (more detailed) guarantees that extend beyond schedulability.

Researchers have proposed alternate richer guarantees. In [4], the authors proposed two metrics on the deadline hit and miss patterns, which they called $\mu$-patterns. A $\binom{n}{m}$ $\mu$-pattern has at least $n$ deadline hits within $m$ consecutive jobs, and a $\binom{n}{m}$ $\mu$-pattern has at least $n$ consecutive deadline hits within any $m$ consecutive jobs. These patterns can be generalized by the notion of regular languages as proposed in [3]. A slightly different approach was followed in [5]. The authors propose a dual guarantee. A nominal guarantee is the typical schedulability test, while an exceptional guarantee specifies for how long deadlines can be missed after an exceptional event.

For the above, a major challenge is the computation or verification of the guarantees. The $\mu$-patterns approach has been demonstrated for fixed-priority scheduling in [4]. The use of regular languages has been shown for time-triggered architectures in [6]. In [5], the settling-time approach is applied when modeling in Real-Time Calculus [7]. It is not clear how to compute these guarantees in a general setting.

Like in [3], we use a regular language that models the patterns of deadlines hits and misses as a guarantee. However, we focus on computing the guarantee for a broad class of scheduling algorithms and task models. To this end, we first represent the real-time system by a network of timed automata [8] in the model-checking tool UPPAAL [9]. Then, we aim to compute a language-based guarantee by model-checking for different candidate languages until we find the one that correctly represents the observed hit and miss patterns. Thus, our approach is limited only by the ability to model a real-time system as a network of timed automata and by the computational cost of model-checking.

In spite of their generic applicability, a common limitation of model-checking tools is the state-space explosion and the consequent high computation cost. Being conscious of this, we make two specific choices in our model-checking process. First, we use a structured approach to identify candidate languages to model-check. To this end, we employ an iterative procedure to incrementally compute a language of a given complexity that models the observed deadline hit and miss patterns. This is similar to the counterexample guided abstraction refinement (CEGAR) [10] approach proposed for program verification. Second, we gradually increase the complexity of the language we wish to use as the guarantee, while fully re-using the information gathered from earlier steps. Here, we define an uncertainty metric to decide when to terminate the iterative process of increasing the language complexity. With these two steps we propose a standard approach to compute the language-based guarantee.

We present two different applications of the language-based guarantee. First, using language inclusion, we demonstrate whether the guaranteed real-time performance matches the assumed performance by another sub-system. As an example, we choose the controller performance of an inverted pendulum with a Linear Quadratic Regulator. For chosen system parameters, the controller performance cannot be ascertained by the schedulability guarantee, but is ascertained by the language-based guarantee. In the second application, we guide the design
of mixed-criticality systems [11] to ‘fairly’ distribute resources among low-criticality tasks in exceptional cases when high-criticality tasks require more resources than usual. We do this using a worst-case deadline miss rate metric. Such a design step is usually not supported by schedulability analysis.

The rest of the paper is organized as follows. In Section II we discuss the model of the real-time system and formally define the guarantee we propose to compute. In Sections III and IV we formally describe the approach to compute the language-based guarantee. In Section V we present two applications of the language-based guarantee and illustrate them with numerical examples.

II. SYSTEM MODEL

In this section, we will describe the model of the real-time system, define the language-based guarantee we want to compute, and illustrate these with an example.

A. Real-Time System

We consider a real-time system with a task-set, \( T = \{\tau_0, \tau_1, ..., \tau_n\} \), running on a single or multiple processors with blocking access to shared resources. We compute the language-based guarantee (which characterizes the deadline hit and miss patterns of jobs) of a specific task, say \( \tau_i \).

The only requirement of our approach is that the timing behavior of such a system be modeled by a network of timed automata [8], which we denote as \( S \) (for system). Thus, our approach is restricted only by the modeling power of timed automata and the computational cost of model-checking.

B. Guarantee Language \( L_G \)

All observed deadline hit and miss patterns of the jobs of a task \( \tau \) can be thought of as strings in a language, denoted as \( L_S \) (for the system language). \( L_S \) is over the alphabet \( \Sigma = \{H, M\} \), where \( H \) implies a hit event and \( M \) implies a miss event. An example string is \( (MHH)^* \), which is a periodic pattern with a deadline miss followed by two deadline hits.

We aim to identify a guarantee that closely, yet conservatively, approximates \( L_S \). To this end, we compute a regular language \( [3] \), denoted \( L_G \) (for the guarantee language), over the alphabet \( \Sigma = \{H, M\} \). As we will see later, a regular language admits advantages which will be useful in the model-checking. We say \( L_G \) is a correct guarantee if

\[
L_S \subseteq L_G. \tag{1}
\]

If a property is satisfied for all strings (or no string) of a correct guarantee \( L_G \), then it follows that it is (or is not) satisfied on \( L_S \). An example of such a property is: “Every deadline miss is followed by at least two deadline hits”.

A regular language can be represented by an equivalent Deterministic Finite Automaton (DFA). In a DFA, every state encodes a finite observed history. Let \( k \) denote the length of the longest history encoded by any state of a DFA, then we represent the DFA as \( A_k \). The parameter \( k \) controls the complexity of the corresponding language: a larger \( k \) corresponds to a larger automaton and possibly a more accurate guarantee.

For the \( A_k \) accepting our guarantee language, denoted \( L_G(k) \), the history encoded in the states is the pattern of deadline hits and misses of the last \( k \) jobs of task \( \tau \). Then, given a hit and miss pattern of the last \( k \) jobs, \( A_k \) can be used to determine if such a pattern can be observed in \( L_S \), and if so, whether the next job would meet or miss its deadline. This is a Markov interpretation, where only the last \( k \) hits and misses can influence the next job’s deadline hit or miss. Similar ideas have been proposed in the analysis of cache hits and branch predictions [12].

**Example 1:** Consider a task-set \( T \) with three periodic tasks \( \tau_0, \tau_1, \tau_2 \in T \) running on three separate cores and accessing a shared memory. Each task has three distinct phases: read data from memory, execute on respective cores using the read data, and write modified data back to memory. Such phased models of tasks have been shown to model many practical applications in the control and real-time domain [13]. The period and the minimum and maximum time units for each phase of each task (as computed, e.g. with static analysis,) are shown in Table I. Contention time before read or write is not included. The arbitration on the memory follows a non-preemptive first-come-first-serve policy.

We examine \( \tau_0 \) for the deadline of \( D_0 = 10 \). Contention and blocking accesses to shared memory lead to variable response times of jobs of \( \tau_0 \) resulting in the deadline hit and miss patterns. Using the technique described in the next sections, we derive the language-based guarantee shown in Table II. This language corresponds to a DFA \( A_k \) with \( k = 2 \), i.e., given the deadline hit and miss pattern of the last two jobs, the language determines the guarantee for the next job. For instance, if \( \tau_0 \) encounters a deadline hit followed by a miss, i.e. \( HM \), then the next event can be either a hit or miss. But, for two consecutive misses, i.e. \( MM \), the next event is guaranteed to be a hit.

III. CONSTRUCTING \( A_k \) FOR A GIVEN \( k \)

In this section, we will describe how we construct the DFA \( A_k \) for a given value of \( k \), such that the equivalent regular language \( L_G(k) \) conservatively approximates \( L_S \). We will first describe a specific template for \( A_k \) which can be iteratively modified. We will then present an observer timed automaton which models our current estimate of \( A_k \), denoted as \( A_k^i \) for the \( i^{th} \) iteration. We then verify a property to check if the estimated guarantee language conservatively approximates \( L_S \). If a counterexample is generated, we show how to modify the observer, or equivalently construct \( A_k^{i+1} \) from \( A_k^i \). Finally, when no more counterexamples are generated and the iterative procedure terminates, we obtain \( A_k \) and show properties of the corresponding \( L_G(k) \).

A. The Observer Automaton

We begin by defining a template for \( A_k \). The DFA \( A_k \) is given by a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where \( Q \) is the set of

<table>
<thead>
<tr>
<th>TASK MODEL FOR EXAMPLE 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>( \tau_0 )</td>
</tr>
<tr>
<td>( \tau_1 )</td>
</tr>
<tr>
<td>( \tau_2 )</td>
</tr>
</tbody>
</table>

**TABLE I.**

**TABLE II.**

**Guarantee Language \( L_G(2) \) for Example 1**

<table>
<thead>
<tr>
<th>Last Two Events</th>
<th>HH</th>
<th>HM</th>
<th>MH</th>
<th>MM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Next Event</strong></td>
<td>H or M</td>
<td>H or M</td>
<td>H or M</td>
<td>H</td>
</tr>
</tbody>
</table>

| **Example 1:** Consider a task-set \( T \) with three periodic tasks \( \tau_0, \tau_1, \tau_2 \in T \) running on three separate cores and accessing a shared memory. Each task has three distinct phases: read data from memory, execute on respective cores using the read data, and write modified data back to memory. Such phased models of tasks have been shown to model many practical applications in the control and real-time domain [13]. The period and the minimum and maximum time units for each phase of each task (as computed, e.g. with static analysis,) are shown in Table I. Contention time before read or write is not included. The arbitration on the memory follows a non-preemptive first-come-first-serve policy.

We examine \( \tau_0 \) for the deadline of \( D_0 = 10 \). Contention and blocking accesses to shared memory lead to variable response times of jobs of \( \tau_0 \) resulting in the deadline hit and miss patterns. Using the technique described in the next sections, we derive the language-based guarantee shown in Table II. This language corresponds to a DFA \( A_k \) with \( k = 2 \), i.e., given the deadline hit and miss pattern of the last two jobs, the language determines the guarantee for the next job. For instance, if \( \tau_0 \) encounters a deadline hit followed by a miss, i.e. \( HM \), then the next event can be either a hit or miss. But, for two consecutive misses, i.e. \( MM \), the next event is guaranteed to be a hit.

**III. CONSTRUCTING \( A_k \) FOR A GIVEN \( k \)**

In this section, we will describe how we construct the DFA \( A_k \) for a given value of \( k \), such that the equivalent regular language \( L_G(k) \) conservatively approximates \( L_S \). We will first describe a specific template for \( A_k \) which can be iteratively modified. We will then present an observer timed automaton which models our current estimate of \( A_k \), denoted as \( A_k^i \) for the \( i^{th} \) iteration. We then verify a property to check if the estimated guarantee language conservatively approximates \( L_S \). If a counterexample is generated, we show how to modify the observer, or equivalently construct \( A_k^{i+1} \) from \( A_k^i \). Finally, when no more counterexamples are generated and the iterative procedure terminates, we obtain \( A_k \) and show properties of the corresponding \( L_G(k) \).

**A. The Observer Automaton**

We begin by defining a template for \( A_k \). The DFA \( A_k \) is given by a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where \( Q \) is the set of
states, $\Sigma = \{H, M\}$ is the alphabet of accepted inputs, $\delta$ is the transition function defined as $Q \times \Sigma \rightarrow Q$, $q_0 \in Q$ is the start state, and $F$ is the set of accepting states. The following are some properties of $A_k$.

- Each state $q \in Q$ corresponds to a particular string of size up to $k$ over the alphabet $\Sigma$. Thus, there are up to $2^k+1$ states.
- We can represent a state $q$ equivalently by its corresponding string denoted $s(q)$. For example, when the DFA is in a state $q$ with $s(q) = (MHH)$, the previous two jobs have met their deadlines while the one before them missed its deadline. The oldest information is the left-most. For the initial state $q_0$, we have $s(q_0) = \phi$.
- $\delta$ is a partial function, i.e., $\delta(q, x)$ does not have to be defined for every state $q \in Q$ and for both $x \in \Sigma$. If the DFA is in state $q$ and the next input is $x$ where $\delta(q, x)$ is not defined, then the input string is not accepted by $A_k$, and thus does not belong to $L_G(k)$.
- The transition function follows from the string representation of the states. For example, let $q_i, q_j$ be two states with $s(q_i) = (MHH)$ and $s(q_j) = (HHM)$. If $\delta(q_i, M)$ is defined, then it is equal to $q_j$ for $k = 3$.
- $F = \{q \in Q : |s(q)| = k\}$.

Thus, the template for $A_k$ defines a class of automata that accept any regular language with strings of length at least $k$. Any language, which can be represented as shown in Table II, is accepted by a DFA belonging to this class depending on the transitions defined in the DFA.

**Example 2:** For the language in Table II, the corresponding $A_k$ with the above template is shown in Fig. 1b. □

For the template of $A_k$, we design a specific observer, denoted as $O$, as a timed automaton [8] in UPPAAL [9], such that:

- Each state of $A_k$ corresponds to a location of $O$.
- All possible transitions in $A_k$ are modeled in $O$. Every location in $O$ has two outgoing transitions for the two cases of hit and miss events. Transitions not defined in $A_k$ have a variable update $e = 0$ (for enable) in $O$, while the others have a variable update $e = 1$.
- Each transition in $O$ which corresponds to a transition in $A_k$ that accepts an $H$ (similarly $M$) has a synchronization channel receive-event hit? (miss?).

The network of timed automata, $S$, writes to the synchronization channel using send-events hit! and miss! whenever a job of task $\tau$ either meets or misses its deadline, respectively. Thus, for a particular execution trace, as the jobs of $\tau$ meet or miss deadlines, $S$ and $O$ synchronize over corresponding channels. We initialize $O$ such that the variable update rule on each transition is $e = 0$. This corresponds to $A_k^0$ which has no defined transitions.

**B. Iterative Construction**

$S$ is extended to include the observer $O$. Then, (for the extended network of TA,) we verify the following (TCTL) safety property:

$$\forall \square e = 1 \text{ (or equivalently, } A[e == 1 \text{ in UPPAAL})}.$$  \hspace{1cm} (2)

This property asserts that the variable $e$ equals 1 in all states. In terms of our usage of $e$, the property asserts that at all times the transitions taken by the observer are already defined in $A_k^1$. If the property is not satisfied, the model-checker returns a counterexample which is a specific trace of inputs observed in the real-time system but not modeled by the observer. Then, we modify the observer such that the counterexample will not be generated again. The modification is direct: we set $e = 1$ on the transition of $O$ which triggered the counterexample. Equivalently, we define (add) a new transition in $A_k^k$ to generate $A_k^{k+1}$. We then repeat the process of verifying property (2) with the modified observer.

**C. Properties of the Computed $L_G(k)$**

The iterative process will terminate after a finite number of steps, as in each step we define a new transition amongst the finite number of possible transitions of $A_k$. Upon termination, the states in DFA $A_k$ with no incoming transitions are dropped, and $A_k$ and the corresponding $L_G(k)$ satisfy the following:

**Lemma 1 (Correctness):** The language $L_G(k)$ corresponding to the DFA $A_k$ satisfies $L_S \subseteq L_G(k)$.

**Lemma 2 (Tightness):** If $A_k^0$ is obtained by disabling any single transition from $A_k$, then the corresponding language $L_G^0(k)$ does not satisfy $L_S \subseteq L_G(k)$.

The correctness property follows from the fact that asserting property (2) asserts $L_S \subseteq L_G(k)$. The tightness follows from the initialization wherein no transition of $A_k^0$ is defined. Thus, any defined transition in $A_k$ is due to an observed pattern of hits and misses in $S$.

**Example 3:** Consider the setup from Example 1. For $k = 2$, we illustrate the observer in Fig. 1a. The guarantee automaton, shown in Fig. 1b, is then generated using the above procedure. The states are represented as $q_{MH}$, with $y = 0$ for the start state and otherwise equal to the history corresponding to the state. The transition $M$ self-loop on $q_{MM}$ is undefined at the end of the procedure and is thus removed. The computed $L_G(k)$ is shown in Table II. □

**IV. GUARANTEE AUTOMATON FOR UNKNOWN $k$**

The previous section described how to compute $L_G(k)$ for a known parameter $k$. However, for applications in which it is unclear what $k$ should be, we need a method to choose the right $k$. This choice should balance between complexity and accuracy: for a large $k$, $L_G(k)$ can more accurately represent $L_S$, but can be computationally expensive to model-check. In this section, we present an iterative method that sequentially computes $L_G(1), L_G(2), \ldots$. We show how to fully utilize
We will argue the above result in terms of $q_s$ then for $A_k$ state section, and derive $A_k$ k for smaller $k$ cannot have a string that is accepted by $L_e$. For instance, the state $q$ in Fig. 2a. Note that certain transitions are removed from the observer. For instance, the state $q$ with $s(q) = (HMM)$ does not have a $\delta(q, M)$ transition. Also, state $q$ with $s(q) = (MMM)$ is not reachable at all and is thus removed. With this observer and the procedure of the previous section, we compute the guarantee automaton $A_3$ shown in Fig. 2b. □

Example 5: For the running example, the uncertainty metric for different values of $k$ is shown in Table III. Row 3 shows time (in s) required to compute each $A_k$ using the iterative refinement approach (Subsection IV-A). Row 4 shows time (in s) required if each $A_k$ is computed non-iteratively, i.e., without using the iterative refinement procedure. These readings are computed on an Intel® Core™ 2 Quad CPU Q6600 @ 2.40GHz × 4 machine with 4GiB RAM. With increase in $k$, the iterative approach shows a significant performance gain over the non-iterative approach.

C. Scalability

For different representative examples, we computed the guarantees for $k$ from 1 to 4. For FCFS and fixed priority arbitration on the bus, systems with up to five cores could be analyzed within 18 hours on a server-grade Intel® Xeon® @ 2.90Ghz. With TDMA arbitration, each core can be analyzed in isolation. Details of the experiments are in the thesis [14].

V. APPLICATIONS

In this section, we will present applications of the language-based guarantee. First, we will show how to verify a controller’s performance requirement by a language inclusion...
test. Second, we will show how to compare scheduling algorithms for mixed-criticality systems by computing worst-case deadline miss rates for low-criticality tasks.

A. Language Inclusion

In model-based design, a standard step is to check that all assumptions are satisfied by the guarantee. Let \( L_A \) denote the language of hit and miss patterns which meet given performance constraints. Then, the real-time system \( S \) satisfies these constraints if \( L_S \subseteq L_A \). It is sufficient to show that \( L_G \subseteq L_A \), under the correctness property (Lemma 1).

We illustrate language inclusion for the design of a networked control system [1]. Consider an inverted pendulum on a moving cart. Let mass of the cart be 0.5kg, the mass, inertia and length of the pendulum be 0.2kg, 0.006 kg.m² and 0.3m, respectively. Let the coefficient of friction of the cart be 0.1N/m/s. The state of this control plant is given by \( z = [x \ x \ \dot{x} \ \theta \ \dot{\theta}] \), where \( x \) is the displacement of the cart, and \( \theta \) the angular displacement of the pendulum, and derivatives are denoted with a dot on top. With period \( P = 0.6s \), we sense the state and compute a feedback control to change \( \dot{x} \) and \( \dot{\theta} \).

Assuming a sensor-to-actuator delay of exactly one period, we design a Linear Quadratic Regulator (LQR) [15]. Whenever the delay is larger than a period, we do not actuate the controller output and the plant is in an open-loop configuration. Thus, we have a switched linear time-invariant system. We design a Linear Quadratic Regulator (LQR) [15]. Whenever the delay is larger than a period, we do not actuate the control output and the plant is in an open-loop configuration.

Let the given performance requirement be: the augmented state variable \( \dot{x}_0 \) is the controller output and the plant is in an open-loop configuration. Thus, we have a switched linear time-invariant system.

We construct a graph \( G' = (V', E', \rho) \) where the vertices \( V' \) correspond to the states in \( A_k \), the edges \( E' \) are the defined transitions in \( A_k \), and weights

\[
\text{W} \left( \rho \right) = \max_{w \in L_S} \left\{ \frac{|i : w_i = M|}{|w|} \right\}.
\]

From the correctness property (Lemma 1), computing the \( \text{W} \left( \rho \right) \) with \( L_G \) instead of \( L_S \) is a safe over-approximation. Given \( A_k \) corresponding to \( L_G \), we compute \( \text{W} \left( \rho \right) \) as follows:

- We construct a graph \( G' = (V', E', \rho) \) where the vertices \( V' \) correspond to the states in \( A_k \), the edges \( E' \) are the defined transitions in \( A_k \), and weights

\[
G \times \overline{G} = \{(i, \overline{j}) \mid i \sim j \}
\]

is the language complement. We verified that \( \text{W} \left( \rho \right) \) be an infinite length string of deadline hits and misses, where the \( i \)th element \( w_i \) in \( H, M \). The worst-case deadline miss rate \( \text{W} \left( \rho \right) \) is given as

\[
\text{W} \left( \rho \right) = \max_{w \in L_S} \left\{ \frac{|i : w_i = M|}{|w|} \right\}.
\]

The controller is implemented in a distributed real-time system: (a) sensed data is sent via a shared bus to the controller, (b) control output is computed on a dedicated processing unit, (c) control outputs are sent via the same shared bus back to the plant. There are two other tasks which also read data via the bus, compute and write data back through the bus. The

**TABLE IV. TASK PARAMETERS FOR THE CONTROL EXAMPLE**

<table>
<thead>
<tr>
<th>Task</th>
<th>Read Range</th>
<th>Exec Range</th>
<th>Write Range</th>
<th>Period</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_0 )</td>
<td>0.06-0.18</td>
<td>0.06-0.12</td>
<td>0.06-0.12</td>
<td>0.6</td>
</tr>
<tr>
<td>( \tau_1 )</td>
<td>0.9-1.02</td>
<td>0.06-0.12</td>
<td>0.06-0.12</td>
<td>2.4</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>1.38-1.5</td>
<td>0.06-0.12</td>
<td>0.06-0.18</td>
<td>3.6</td>
</tr>
</tbody>
</table>
TABLE V. MIXED CRITICALITY TASK-SET

<table>
<thead>
<tr>
<th>τ</th>
<th>χ</th>
<th>Read</th>
<th>Write</th>
<th>Exec</th>
<th>Period</th>
<th>Deadline</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>HI</td>
<td>5-7</td>
<td>6-9</td>
<td>7-8</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>71</td>
<td>LO</td>
<td>4-6</td>
<td>7-8</td>
<td>5-6</td>
<td>50</td>
<td>30</td>
</tr>
<tr>
<td>72</td>
<td>LO</td>
<td>3-5</td>
<td>7-9</td>
<td>6-7</td>
<td>60</td>
<td>30</td>
</tr>
</tbody>
</table>

We use \( \text{WMR} \) to compare scheduling algorithms for mixed-criticality systems [11]. In mixed-criticality systems, every task \( τ \) has a defined criticality level \( χ_i \), and at any given time, \( t \), there is a global criticality level \( χ_g(t) \). If \( χ_g(t) > χ_i \) then most existing scheduling policies decide to drop task \( τ_i \) from \( t \) onwards. In other words, when the global criticality level rises due to certain exceptional run-time events, low criticality tasks are no more scheduled. We believe this abrupt dropping of tasks is due to the limitation of the schedulability guarantee. Instead, we may still run the low criticality tasks under reduced performance guarantees, in particular with higher \( \text{WMR} \).

Consider a dual-criticality (HI, LO) task-set with three tasks executing on independent processing cores, but interfering on a shared bus which follows a non-preemptive fixed-priority arbitration policy. Tasks read data via the bus, compute using the read data and write data back via the bus. Read and write is done through individual accesses, each requiring 1 time unit once access to the bus is granted. The ranges of read-write access requests and execution times (in time units) for the different tasks, when the global criticality level is HI, are shown in Table V. In this case, tasks \( τ_1 \) and \( τ_2 \) may not be guaranteed to meet all their deadlines. But we can guarantee a certain \( \text{WMR} \) for each.

We consider two priority assignments, \( \text{PA}_1 = τ_0 > τ_1 > τ_2 \) and \( \text{PA}_2 = τ_0 > τ_2 > τ_1 \). For both, we compute the language-based guarantee for \( τ_1 \) and \( τ_2 \). We do not have a pre-defined \( k \) so we use our iterative refinement procedure to increase \( k \) with the terminating condition: \( U(k) < 0.1 \) or \( \frac{U_{k+1}}{U_k} > 0.9 \). We then compute \( \text{WMR} \) for each task of both priority assignments. The results in Table VI are interpreted as follows:

- Fairness amongst LO-criticality tasks: The absolute value of the difference between \( \text{WMR} \) of \( τ_1 \) and \( τ_2 \) is smaller for \( \text{PA}_2 \).
- Responsiveness of LO-criticality tasks: The sum of the \( \text{WMR} \) of \( τ_1 \) and \( τ_2 \) is higher for \( \text{PA}_2 \).

Hence, \( \text{PA}_2 \) is clearly a better priority assignment. However, when using only the schedulability guarantee, the two priority assignments are indistinguishable in the HI-criticality mode, and both \( τ_1 \) and \( τ_2 \) will be dropped in either case. We can similarly compare two different scheduling policies using the language-based guarantee.

TABLE VI. MIXED CRITICALITY RESULTS

<table>
<thead>
<tr>
<th>k</th>
<th>( \text{WMR} )</th>
<th>( \text{PA}_1 )</th>
<th>( \text{PA}_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>0.3125</td>
<td>0.75</td>
<td>0.5415</td>
</tr>
</tbody>
</table>

\( \rho : E \to \{-1, 1\} \) is -1 (similarly 1) on edges if the corresponding transition accepts \( M(H) \).

- We compute the minimum cycle mean of \( G' \), denoted MCM, with the Minimum Cycle Mean algorithm [17].
- Then, \( \text{WMR} = \frac{1 - \text{MCM}}{2} \).

VI. Conclusion

We proposed a language-based guarantee that details the deadline hit and miss patterns of the jobs of a task, then provided a method to compute this guarantee and finally presented two applications to describe its applicability. We also showed the superiority of the language-based guarantee against the schedulability guarantee. With a wide scope of application, the only limitation of our method is the inevitable cost of model-checking. Yet, we have reduced such costs by using a suitable observer template and corresponding modification procedure, and an iterative refinement approach that reuses information from one iteration to the next.

In conclusion, we believe the language-based guarantee is a viable option for the design and analysis of cyber-physical systems. We are interested to study the implications of the detailed guarantee in other application domains.

Acknowledgement: This work was partially funded by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 288175.

REFERENCES