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 modelled 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 μ-patterns. A $\binom{n}{m}$ μ-pattern has at least n deadline hits within m consecutive jobs, and a $\left\langle {n \atop m} \right\rangle$ μ-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 μ-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 reusing 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

978-3-9815370-2-4/DATE14/©2014 EDAA

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 τ_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 τ 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 τ. 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].

TABLE I. Task model for Example 1

Task	Read	Execute	Write	Period
τ_0	1-2	1-3	1-2	15
τ_1	1-2	10-14	1-2	40
τ_2	1-3	12-15	1-3	50

TABLE II. Guarantee language $L_G(2)$ for Example 1

Last Two Events	HH	HM	MH	MM
Next Event	H or M	H or M	H or M	H

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 τ_0 for the deadline of $D_0 = 10$. Contention and blocking accesses to shared memory lead to variable response times of jobs of τ_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 τ_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, δ is the transition function defined as $Q \times \Sigma \to Q$, $q_0 \in Q$ 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 Σ. Thus, there are up to $2^{k+1} - 1$ 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 $s(q_0) = \phi$.
- δ 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:

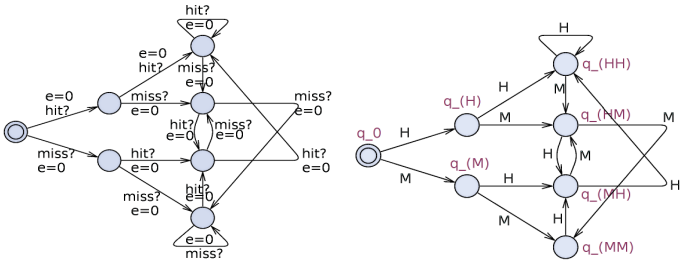
- Each state of A_k corresponds to a location of O.
- All possible transitions in A_k are modeled in O. Every location in O has two outgoing transitions for the two cases of hit and miss events. Transitions not defined in A_k have a variable update $e = 0$ (for enable) in O, while the others have a variable 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 τ either meets or misses its deadline, respectively. Thus, for a particular execution trace, as the jobs of τ 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:

$$\forall\Box e = 1 \text{ (or equivalently, } A[]e == 1 \text{ in UPPAAL).} \quad (2)$$



(a) Initial Observer O for $k = 2$. (b) Guarantee Automaton A_2.

Fig. 1. Refinement Illustration.

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^i. 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^i to generate A_k^{i+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 $L_S \subseteq L_G(k)$.

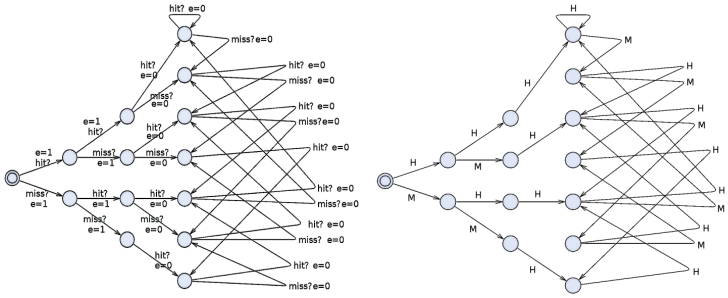
Lemma 2 (Tightness): If A'_k is obtained by disabling any single transition from A_k, then the corresponding language $L'_G(k)$ does *not* satisfy $L_S \subseteq L'_G(k)$.

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_y, 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



(a) Initial Observer for $k = 3$ from A_2. (b) Guarantee Automaton A_3.

Fig. 2. Refinement Illustration for Unknown k.

$L_G(k)$ when computing $L_G(k+1)$, propose a terminating condition for this method, and address scalability issues.

The following lemma forms the basis of our approach. It states that the computed language (as in the previous section) for a smaller k cannot be a tighter approximation of L_S.

Lemma 3 (Inclusion): $L_G(k+1) \subseteq L_G(k)$, $\forall k > 0$.

We will argue the above result in terms of A_k and A_{k+1}. For every state q in A_{k+1} we define a *parent state* $p(q)$ in A_k. Two cases arise based on the string representation $s(q)$. If $|s(q)| < k+1$, then $s(p(q)) = s(q)$. If $|s(q)| = k+1$, then $s(p(q))$ is obtained by dropping the left-most (oldest) element in $s(q)$. This definition is such that, when processing a common input, if A_{k+1} is in state q, then A_k has to be in state $p(q)$. From the tightness of A_{k+1} and the correctness of A_k, we have: if either or both transition(s) (out of H and M) are disabled in A_k for state $p(q)$, then the corresponding transitions are also disabled in A_{k+1} for state q. Thus, we cannot have a string that is accepted by A_{k+1} and not by A_k.

A. Refining $L_G(k)$ by Increasing k

We provide an iterative refinement approach for increasing k till we get a satisfactory $L_G(k)$. We start with the observer for $k = 1$, modify it using the procedure in the previous section, and derive A_1. Then, we begin the process for A_2. However, we can now use information from A_1 to modify A_2^0 before iterative verification with the model-checker. We know that for every state q in A_2^0, if the parent state $p(q)$ in A_1 has either or both transition(s) disabled, then the corresponding transitions will also be disabled in A_2. Using this we modify O, and hence A_2^0, by removing certain transitions. The modified observer is now used to obtain A_2. This process of constructing A_{k+1} by using information from A_k is termed *refinement*.

Example 4: Consider guarantee automaton A_2 shown in Fig. 1b. Using this we initialize the observer for $k = 3$, shown 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. □

This process is complete in the sense that any counterexample generated when computing A_k will not be generated when computing A_{k+1}. This reduces the model-checking time.

TABLE III. UNCERTAINTY METRIC FOR EXAMPLE 1

k	1	2	3	4	5	6	7	8
$U(k)$	1.000	0.929	0.733	0.500	0.317	0.197	0.120	0.072
Iterative	0.949	2.219	4.335	6.798	9.695	13.901	19.401	27.635
Non-iterative	0.948	2.276	4.750	7.800	11.824	18.438	29.786	53.339

B. Terminating Condition

We propose *Uncertainty Metric*, denoted $U(k)$, to define a terminating condition for the iterative process of increasing k

$$U(k) = \frac{e_{pr}}{e_{po}}, \tag{3}$$

where e_{pr} is the number of transitions present in A_k, and $e_{po} = 2^{k+2}$, the total number of transitions possible in A_k. The metric conveys the level of uncertainty in A_k with respect to the predictions at each state, and is similar to the statistical metric of entropy.

$U(k)$ varies between 0 and 1. $U(k) = 1$ implies that every state in A_k has two outgoing transitions. Thus, irrespective of the previous sequence of hits and misses, the next event can either be a hit or miss. A lower value of $U(k)$ implies that there is lesser uncertainty, because there are states with either one or no outgoing transitions.

From Lemma 3, it is clear that A_{k+1} will have a maximum of twice the number of transitions as A_k. e_{po} increases to twice the value for each increase in k. Hence, the ratio either remains constant or decreases with an increase in k, i.e.,

Lemma 4: $U(k+1) \leq U(k)$.

Using the above property, we can decide to terminate the iterative process if $U(k)$ reaches a value close to 0. However, for some settings, uncertainty may be inevitable. For instance, if every job of τ can either hit or miss its deadline, then $U(k) = 1$ irrespective of k. Thus, in addition to the absolute terminating condition, we also have a relative condition, such as $U(k+1)/U(k)$ is a value close to 1.

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.m^2 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 $z = [x\ \dot{x}\ \theta\ \dot{\theta}]$, where x is the displacement of the cart, and θ the angular displacement of the pendulum, and derivatives are denoted with a dot on top. With period $P = 0.6$s, 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 can collate the plant and controller state variables at the n^{th} sampling time by $X[n]$ with the following dynamics

$$X[n+1] = A_{cl}X[n], \quad \text{if } d[n] \leq P,$$
$$X[n+1] = A_{ol}X[n], \quad \text{otherwise},$$

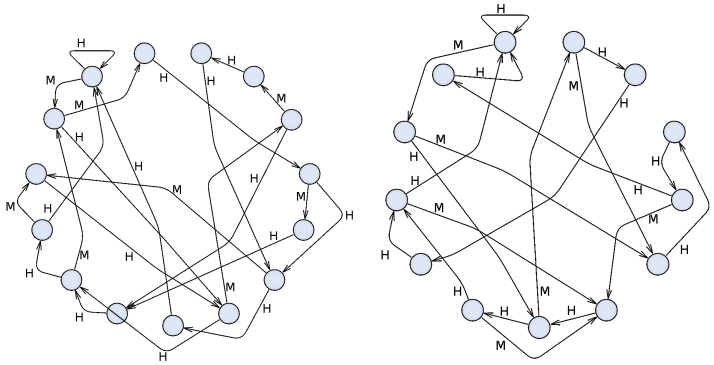
where A_{cl} and A_{ol} are the system matrices for closed-loop and open-loop, respectively, $d[n]$ is the sensor-to-actuator delay for the n^{th} period, and P is the sampling period. For the computed LQR controller, we have $\|A_{cl}\| = 0.82$, and $\|A_{ol}\| = 28.2$.

Let the given performance requirement be: the augmented state variable X must be exponentially stable such that

$$\frac{\|X[n+6]\|}{\|X[n]\|} < 0.5, \quad \forall\ n > 0, X[n]. \tag{4}$$

This condition specifies that the *energy* in the vector X must at least halve in every 6 periods. This is a stronger condition than asymptotic stability. This condition may be satisfied for different deadline hit and miss patterns of 6 consecutive control signals. Let a such a pattern by represented as $\sigma = (\sigma_1, \ldots, \sigma_6)$, where $\sigma_i \in \{H, M\}$. For a given pattern, there is a corresponding matrix $A_\sigma = \Pi_{i=1,\ldots,6} A_i$, where $A_i = A_{cl}$ if $\sigma_i = H$ and $A_i = A_{ol}$ if $\sigma_i = M$. As shown in [3], condition of (4) is satisfied for a pattern σ if the corresponding matrix A_σ has an eigenvalue less than 0.5. All patterns satisfying this condition are specified as a regular language L_A. The minimized automaton corresponding to L_A for the computed matrices A_{cl} and A_{ol} is shown in Fig. 3a.

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 arbitration policy on the bus is non-preemptive first-come-first-serve. Thus, the bus is a shared resource which can increase the sensor-to-actuator delay. The minimum and maximum time (in s), for each phase of each task is shown in Table IV. Contention time before read or write is not included. τ_0 is the controller task for the inverted pendulum with a period of 0.6s.



(a) Minimized Automaton for L_A. (b) Minimized Automaton for L_G.

Fig. 3. The Minimized Automata.

TABLE IV. TASK PARAMETERS FOR THE CONTROL EXAMPLE

Task	Read Range	Exec Range	Write Range	Period
τ_0	0.06-0.18	0.06-0.12	0.06-0.12	0.6
τ_1	0.9-1.02	0.06-0.12	0.06-0.12	2.4
τ_2	1.38-1.5	0.06-0.12	0.06-0.18	3.6

Applying the method of [16], the worst-case response time of τ_0 is 0.72s $> P$. Using the schedulability guarantee, the plant can never be guaranteed to be in the closed-loop mode, and thus is not guaranteed to meet the requirement of (4). Further, no amount of speed-up of the processor executing τ_0 can meet the controller requirement. One would have to speed up the bus by at least 37.5% to satisfy the requirement.

Since the given L_A accepts strings of size 6, we compute L_G for $k = 5$. Fig. 3b shows the corresponding minimized automaton. We then compute $L_{\texttt{diff}} = L_G \cap \neg L_A$, where $\neg L_A$ is the language complement. We verified that $L_{\texttt{diff}} = \phi$, and thus $L_G \subseteq L_A$. This shows that the real-time system can meet the controller constraint of (4), contrary to the conclusion from the schedulability guarantee. More details are in [14].

B. Calculating Worst-Case Deadline Miss Rate

While the language-based guarantee is very detailed, it can be used to compute specific metrics. An example of this is the worst-case deadline miss rate. Let w 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 WMR is given as

$$\texttt{WMR} = \max_{w \in L_S} \frac{|\{i\ :\ w_i = M\}|}{|w|}. \tag{5}$$

From the correctness property (Lemma 1), computing the WMR with L_G, instead of L_S, is a safe over-approximation.

Given A_k corresponding to L_G, we compute WMR 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

TABLE V. MIXED CRITICALITY TASK-SET

τ_i	χ_i	Read	Write	Exec	Period	Deadline
τ_0	HI	5-7	6-9	7-8	80	80
τ_1	LO	4-6	7-8	5-6	50	30
τ_2	LO	3-5	7-9	6-7	60	30

TABLE VI. MIXED CRITICALITY RESULTS

	$\texttt{PA}_1$		$\texttt{PA}_2$	
	τ_1	τ_2	τ_1	τ_2
k	9	12	15	8
WMR	0.3125	0.75	0.5415	0.25

$\rho : E \rightarrow \{-1, 1\}$ 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, $\texttt{WMR} = \frac{1-\texttt{MCM}}{2}$.

We use WMR to compare scheduling algorithms for mixed-criticality systems [11]. In mixed-criticality systems, every task τ_i has a defined criticality level say χ_i, and at any given time, t, there is a global criticality level say $\chi_g(t)$. If $\chi_g(t) > \chi_i$ 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 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 WMR for each.

We consider two priority assignments, $\texttt{PA}_1 = \tau_0 > \tau_1 > \tau_2$ and $\texttt{PA}_2 = \tau_0 > \tau_2 > \tau_1$. 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 $U_{k+1}/U_k > 0.9$. We then compute 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 WMR of τ_1 and τ_2 is smaller for $\texttt{PA}_2$.
- Responsiveness of LO-criticality tasks: The sum of the WMR of τ_1 and τ_2 is higher for $\texttt{PA}_2$.

Hence, $\texttt{PA}_2$ 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.

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

[1] A. Bemporad, M. Heemels, and M. Johansson, *Networked control systems*. Springer, 2010, vol. 406.

[2] J. P. Hespanha, P. Naghshtabrizi, and Y. Xu, "A survey of recent results in networked control systems," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 138–162, 2007.

[3] R. Alur and G. Weiss, "Regular specifications of resource requirements for embedded control software," in *RTAS*, 2008, pp. 159–168.

[4] G. Bernat, A. Burns, and A. Llamosí, "Weakly hard real-time systems," *IEEE Trans. Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[5] P. Kumar and L. Thiele, "Quantifying the effect of rare timing events with settling-time and overshoot," in *RTSS*, 2012, pp. 149–160.

[6] A. D'Innocenzo, G. Weiss, R. Alur, A. J. Isaksson, K. H. Johansson, and G. J. Pappas, "Scalable scheduling algorithms for wireless networked control systems," in *CASE*, 2009, pp. 409–414.

[7] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *ISCAS*, 2000, pp. 101–104.

[8] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[9] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer aided verification*. Springer, 2000, pp. 154–169.

[11] S. Baruah, H. Li, and L. Stougie, "Towards the design of certifiable mixed-criticality systems," in *RTAS*, 2010, pp. 13–22.

[12] I. Hur and C. Lin, "Adaptive history-based memory schedulers," in *MICRO*, 2004, pp. 343–354.

[13] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley, "A predictable execution model for cots-based embedded systems," in *RTAS*, 2011, pp. 269–279.

[14] N. Dhruva, "Crossing the Deadline: An Automata-Based Hard Real-Time Guarantee," Bachelor's Thesis, ETH Zurich, 2013. [Online]. Available: ftp://ftp.tik.ee.ethz.ch/pub/students/2013-HS/BA-2013-17.pdf

[15] J. Hespanha, "Lecture notes on lqr/lqg controller design." [Online]. Available: http://www.uz.zgora.pl/ wpaszke/materialy/kss/lqrnotes.pdf

[16] G. Giannopoulou, K. Lampka, N. Stoimenov, and L. Thiele, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *EMSOFT*, 2012, pp. 63–72.

[17] R. M. Karp, "A characterization of the minimum cycle mean in a digraph," *Discrete Mathematics*, vol. 23, no. 3, pp. 309 – 311, 1978.