QOC-ORIENTED EFFICIENT SCHEDULE SYNTHESIS FOR MIXED-CRITICALITY CYBER-PHYSICAL SYSTEMS

Reinhard Schneider, Dip Goswami, Alejandro Masrur, Samarjit Chakraborty

Institute for Real-Time Computer Systems, TU Munich, Germany

ABSTRACT

Cyber-physical systems (CPS) are characterized by a tight interaction between computational resources and physical systems. Such systems typically consist of a mix of time-critical real-time tasks and safety-critical control tasks. Time-critical applications are normally associated with hard real-time constraints which need to be guaranteed by the system. On the other hand, control applications are not strictly related to deadlines but rather to quality of control (QoC). Traditional scheduling policies such as Deadline Monotonic can guarantee timing constraints, however, they do not allow for QoC optimized schedules. Optimizing overall QoC while guaranteeing all deadlines constitutes a challenging scheduling problem which is increasingly attracting attention. In this paper, we present an efficient schedule synthesis algorithm for such *mixed-criticality systems*. The proposed algorithm has a polynomial complexity and ensures all hard real-time constraints while maximizing overall QoC for all control applications.

Index Terms— Scheduling algorithm, cyber-physical systems, mixed-criticality systems, real-time systems

I. INTRODUCTION

Mixed-criticality systems are built up of systems implementing different levels of criticality. Such safety-critical systems are present in various application domains and are typically certified according to stringent industrial standards. For instance:

- the DO-178B [16] specification is an avionics software standard which describes processes for development and analysis of software used in airborne systems.
- ISO 26262 [14] is an automotive safety standard, titled *"Road vehicles – Functional safety"* which covers functional safety aspects of the entire development process including specification and verification of software safety requirements, and software architectural design.
- IEC 61508 [12] is a fundamental industrial safety standard which describes requirements for functional safety of electronic safety-related systems.

In this context, we consider *time-critical real-time tasks* and *safety-critical control tasks* which constitute a *mixed criticality system*. The system realized by time-critical real-time tasks depends on hard real-time constraints, *deadlines*, that need to be satisfied under all circumstances. Thus, any deadline miss can result in a malfunction of the system. As a result, guaranteeing deadlines is a *necessary* and *sufficient* condition to ensure a flawless behavior of such systems.

In contrast, system stability is a *necessary* condition which must be realized by safety-critical control tasks implementing the functionality. However, in most cases stability is not a *sufficient* condition to ensure the desired system behavior. This is because the realized functionality is qualified according to the control performance (QoC), e.g., it is important that the system follows a certain trajectory. Hence, optimized QoC results in another design objective which needs to be carefully considered at design time.

In many application domains such as in the automotive industry, software of different levels of criticality, coexist on the same platform and share processing resources. Often, the goal is to achieve a segregation of critical and non-critical functions in order to ensure freedom of interference and to guarantee safe system behavior. However, this involves specialized hardware and software effort which in turn may reduce flexibility and increase costs, and hence more integrated approaches are desired. Consequently, the integration of software of mixed-criticality to optimize costs, weight, and energy consumption gives rise to a challenging scheduling problem that we want to tackle in this work.

Related work: Of late there has been tremendous research effort towards analysis and synthesis of mixed-criticality systems. The work in [8] proposes a dependability driven approach for the integration of both safety-critical and non-safety critical software functionalities on shared resources. In this context, the mapping process considers functional and extra-functional constraints of dependability and real-time.

Recently there has been research interest in schedule synthesis for mixed-criticality systems. Baruah et al. [1], [3], introduce formal models for mixed-criticality workloads and present a response-time analysis method for fixed priority uniprocessor scheduling in mixed criticality systems. Further, [9] and [7] describe schedule synthesis techniques in the context of mixed criticality systems. The above lines of work address mixed criticality in the sense of jointly scheduling hard real-time tasks and soft real-time tasks. However, they do not consider control tasks. Further there has been recent work on schedule synthesis for control applications with the goal to maximize QoC [15], [17], [18]. These approaches made significant contributions to controler/scheduler co-design, however, they did not consider the joint schedule synthesis of real-time tasks and control tasks in a mixed-criticality environment.

Problem statement and overview of our scheme: Our goal is to automatically synthesize schedules for a set of real-time tasks τ_{rt} and a set of control tasks τ_c that are executed on a shared resource. As the underlying scheduling policy we consider fixed priority preemptive scheduling (FPPS), and hence, the schedule synthesis problem boils down to an offline priority assignment problem. Clearly, it is meaningful to assign priorities to real-time tasks according to the Deadline Monotonic (DM) policy, i.e., the shorter the deadline of a task the higher its priority. This results in an *optimal* priority assignment [10], which means that if a fixed-priority schedule is feasible under any priority assignment, then DM will also be feasible. However, the inverse does not hold, i.e., if DM is feasible, it does not necessarily imply that any other schedule (different to DM) would also be feasible. However note that, even if DM results in an optimal solution for assigning priorities to real-time tasks in the system, it is not applicable to control tasks. In the case of control tasks, an optimal priority assignment depends on the performance of the system realized by such tasks. That is, priorities should be assigned to control tasks in a mixed-criticality environment such that the overall performance is maximized.

In general, a control task's performance is a function of its execution delay: the more execution delay, the worse the performance. However, from the point of view of the performance, different control tasks may behave very differently. Some control tasks are intrinsically more sensitive to execution delay than others. As a result, when synthesizing schedules in a mixed control/real-time task environment, the impact of different priority levels on control tasks needs to be analyzed, and hence, the priority assignment problem becomes much more difficult to solve. Our proposed schedule synthesis scheme is characterized as follows:

- The triggering patterns of tasks are modeled as *arrival curves* which allow for more general characterization of event streams than standard event models. Hence, our scheduling algorithm is not restricted to strictly periodic arrival patterns
- Time-critical real-time tasks are scheduled such that their deadlines are guaranteed in the worst case
- Safety-critical control tasks are scheduled with respect to stability and optimized QoC. For this, we present a heuristic to optimize the overall QoC and compare our results with classical DM scheduling
- We propose a multi-layered scheduling scheme which allows to implement appropriate scheduling strategies for each type of criticality being present in the system in an integrated scheduling framework.

II. THEORETIC BACKGROUND

In this section we give an overview on theoretic background that we use in this work. We first introduce a performance analysis framework followed by a section on control theoretic fundamentals.

II-A. Real-Time Calculus

Real-time calculus (RTC) is an analytical framework for worst-case performance analysis of real-time systems. RTC allows modeling of (i) the triggering pattern of tasks (*event model*) which generate execution demands on a resource, and (ii) the service offered by a resource (*resource model*), e.g., a processor, to each task running on it. The mathematical framework of RTC is based on (min,+) and (max,+) algebra [5].

Terms and definitions: Let $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$ where $\mathbb{R}$ is the set of real numbers and $\mathcal{F}$ be the set of monotonic functions $\mathcal{F} = \{f : \mathbb{R}^+ \to \overline{\mathbb{R}} \mid \forall s < t,\ 0 \leq f(s) \leq f(t)\}$ where $\mathbb{R}^+$ is the set of non-negative real numbers. Further, the *supremum* (sup) of a set $S \subseteq \mathcal{F}$ is the smallest $U \in \mathcal{F}$ such that $h \leq U$ for all $h \in S$. Similarly, the *infimum* (inf) of S is the largest $L \in \mathcal{F}$ such that $h \geq L$ for all $h \in S$.

The (min,+) convolution $\otimes$ and deconvolution $\oslash$ operators are defined as: $\forall f, g \in \mathcal{F}$, $\forall t \in \mathbb{R}^+$

$$\begin{aligned}(f \otimes g)(t) &= \inf\{f(s) + g(t-s) \mid 0 \leq s \leq t\},\\ (f \oslash g)(t) &= \sup\{f(t+u) - g(u) \mid u \geq 0\}.\end{aligned}$$

Similarly, the (max,+) convolution $\overline{\otimes}$ and deconvolution $\overline{\oslash}$ operators are defined as: $\forall f, g \in \mathcal{F}$, $\forall t \in \mathbb{R}^+$

$$\begin{aligned}(f \overline{\otimes} g)(t) &= \sup\{f(s) + g(t-s) \mid 0 \leq s \leq t\},\\ (f \overline{\oslash} g)(t) &= \inf\{f(t+u) - g(u) \mid u \geq 0\}.\end{aligned}$$

The maximum vertical and horizontal deviation (distance) between two functions $f, g \in \mathcal{F}$ are given by:

$$\mathsf{vdist}(f,g) \overset{\text{def}}{=} \sup\{\ f(t) - g(t) \mid t \geq 0\ \} \tag{1}$$

$$\mathsf{hdist}(f,g) \overset{\text{def}}{=} \sup\{\ \inf\{\tau \geq 0 \mid f(t) \leq g(t+\tau)\} \mid t \geq 0\} \tag{2}$$

Further, a function $f \in \mathcal{F}$ is sub-additive iff $f(x+y) \leq f(x) + f(y)$ for all x and y in $\mathbb{R}^+$. Similarly, f is super-additive iff $f(x+y) \geq f(x) + f(y)$ for all x and y in $\mathbb{R}^+$. In this paper, we assume that all given upper (lower) functions satisfy sub-additivity (super-additivity) before the analysis.

Event model: Data streams are modeled using a *count-based abstraction* where an arrival pattern of a stream is modeled as a cumulative function $R(t)$ denoting the number of events that arrive during the time interval $(0, t]$. The maximum and minimum number of events that are recorded during *any* time interval of length Δ is represented by a pair of *arrival functions* $\alpha = (\alpha^u, \alpha^l)$ that is defined as

$$\forall \Delta \geq 0,\ \forall t \geq 0: \quad \alpha^l(\Delta) \leq R(\Delta + t) - R(t) \leq \alpha^u(\Delta).$$

Arrival curves allow for an expressive characterization of event streams which are able to represent standard event models, e.g., *periodic*, *periodic with jitter* and *sporadic*, as well as arbitrary arrival patterns. Standard event arrival

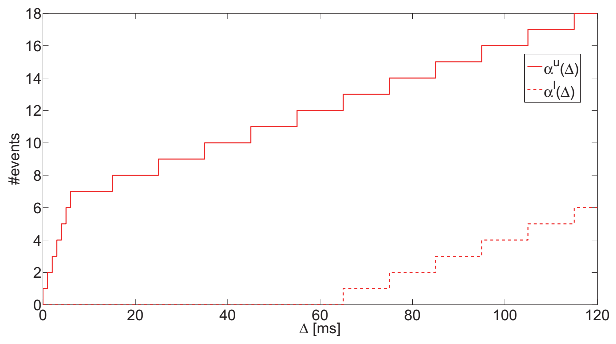

Fig. 1. Arrival curve $\alpha = (10, 55, 1)$

patterns are often specified by the tuple (p, j, d), where p denotes the period, j the jitter, and d the minimum inter-arrival distance of events in the stream. The corresponding pair of arrival curves to this specification is modeled as

$$\alpha^l(\Delta) = \left\lfloor \frac{\Delta - j}{p} \right\rfloor, \text{ and } \alpha^u(\Delta) = \min\left\{ \left\lceil \frac{\Delta + j}{p} \right\rceil, \left\lceil \frac{\Delta}{j} \right\rceil \right\} \tag{3}$$

Fig. 1 shows an example of a pair of arrival curves specified as $\alpha = (10, 55, 1)$, i.e., $p = 10$, $j = 55$, and $d = 1$.

Resource model: Similarly, resource capacities are captured by a cumulative function $C(t)$ denoting the number of events that can be processed by a resource in the time interval $(0, t]$. The maximum and minimum number of events that can be processed in *any* time interval of length Δ is upper- and lower-bounded by a pair of *service functions* $\beta = (\beta^u, \beta^l)$ which is defined as

$$\forall \Delta \geq 0, \ \forall t \geq 0 : \quad \beta^l(\Delta) \leq C(\Delta + t) - C(t) \leq \beta^u(\Delta).$$

Further, β can also be expressed in terms of the maximum and minimum number of available resource units, e.g., processor cycles.

Compositional performance analysis: Let us consider an input data stream which is bounded by the arrival functions $\alpha = (\alpha^u, \alpha^l)$ and processed by task T on a resource with available service $\beta = (\beta^u, \beta^l)$ as illustrated in Fig. 2 a). Further, let us assume the buffer that stores arriving events of the input data stream has infinite capacity.

Then, according to (1) and (2) the maximum backlog B at the input buffer, i.e., the maximum buffer space that is required to buffer this event stream, and the maximum delay D experienced by the input stream α are given by

$$B = \mathsf{vdist}(\alpha^u, \beta^l), \tag{4}$$

and

$$D = \mathsf{hdist}(\alpha^u, \beta^l). \tag{5}$$

The bounds on the output arrival functions α' and remaining service functions β' for a greedy preemptive processing component are computed as follows [6]:

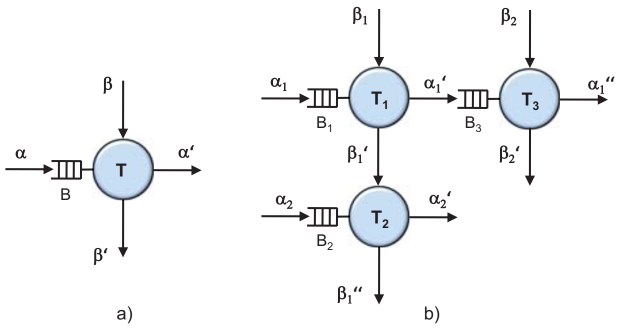

Fig. 2. a) RTC processing model, b) Compositional analysis.

$$\alpha^{u'} = \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \ \beta^u\} \tag{6}$$
$$\alpha^{l'} = \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \ \beta^l\} \tag{7}$$
$$\beta^{u'} = (\beta^u - \alpha^l) \ \overline{\oslash} \ 0 \tag{8}$$
$$\beta^{l'} = (\beta^l - \alpha^u) \ \overline{\otimes} \ 0 \tag{9}$$

Further, the bounds on the processed arrival curves $\alpha' = (\alpha^{u'}, \alpha^{l'})$, and the remaining service $\beta' = (\beta^{u'}, \beta^{l'})$ can be used in a compositional manner. Consider the example in Fig. 2 b) where the input streams α_1 and α_2 are processed by the tasks T_1 and T_2 on a resource with total service β_1. Assume the output of T_1 is processed by task T_3 using service β_2, and T_1 and T_2 are scheduled according to FPPS where T_1 is assigned a higher priory than T_2. Then, the full service β_1 is available to the task with the highest priority (T_1) to process its input stream (α_1). The backlog and worst-case delay are computed as $B_1 = \mathsf{vdist}(\alpha_1^u, \beta_1^l)$ and $D_1 = \mathsf{hdist}(\alpha_1^u, \beta_1^l)$. The bounds on the processed output stream $\alpha_1{}'$ and the remaining service $\beta_1{}'$ are computed using (6)-(9). Now, $\beta_1{}'$ is used to compute the performance bounds of task T_2 which processes input stream α_2, i.e., $B_2 = \mathsf{vdist}(\alpha_2^u, \beta_1^{l'})$ and $D_2 = \mathsf{hdist}(\alpha_2^u, \beta_1^{l'})$. In a similar way, the processed output stream $\alpha_1{}'$ is used as an input for task T_3 using service β_2.

II-B. Control Theory

A feedback control system aims to achieve the desired behavior of a dynamical system by applying appropriate inputs (that is computed based on the feedback signals) to the system. In general a dynamical system is modeled by a set of differential equations called the *state-space model*,

$$\dot{x}(t) = Ax(t) + Bu(t), \tag{10}$$

where $x(t) \in R^n$ is the system *state* and $u(t) \in R$ is the *control input* to the system. $A \in R^{n \times n}$ and $B \in R^{n \times 1}$ are the system and input matrices, respectively. *State-feedback control* essentially implies the design of $u(t)$ as a function of the states $x(t)$ (feedback signals) so as to meet certain high-level design requirements. A feedback control loop performs mainly three operations:

- measure the states $x(t)$ (*measure*),
- compute input signal $u(t)$ (*compute*) and,
- apply the computed $u(t)$ to the plant (10) (*actuate*).

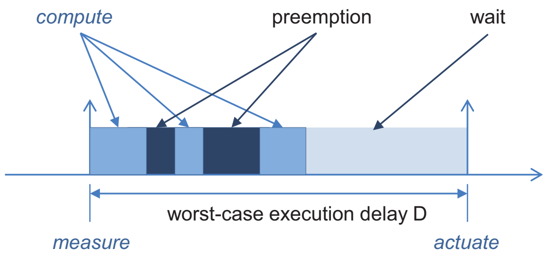

Fig. 3. Control task model

Performing these operations in a continuous fashion in any implementation platform requires infinite computation power. Hence, in a digital implementation platform of such feedback loop, these operations are performed only at discrete-time intervals (sampling instants) t_k for $k \in \mathbb{N}_0$. The time interval between two consecutive executions of a feedback control loop (*control task*) is the sampling period $t_{k+1} - t_k = h$. In this work, we consider state-feedback controllers of the form $u(t) = Kx(t)$ where K is the state-feedback gain which needs to be designed. The time duration from the measure operation to the actuate is known as *sensor-to-actuator* delay or *response time* D of a control task. For the rest of the paper, we will refer to D as *execution delay* of a control task $T \in \tau_c$ as depicted in Fig. 3. We assume a control task model where the operations *measure*, *compute*, and *actuate* are performed in one task. Further, we consider a task model where the actuation is performed after worst-case delay D even if the computation of the control input is completed earlier (see Fig. 3) to avoid jitter in the execution delays of the control tasks.

In the process of executing such a control loop, the control input $u(t)$ is held until the next update comes, i.e.,

$$u(t) = Kx(t_k), \quad t_k \leq t \leq t_{k+1} \tag{11}$$

Given the input signal (11) and the execution delay being $D < h$, the continuous-time system (10) becomes a *sampled-data* system [4],

$$x[k+1] = A_d x[k] + B_0(D)u[k] + B_1(D)u[k-1], \tag{12}$$

where

$$A_d = e^{Ah},\ B_0(D) = \int_0^{h-D} e^{At}dt \cdot B,\ B_1(D) = \int_0^{D} e^{At}dt \cdot B$$

Putting (11) in (12), we get the following *closed-loop system*,

$$x[k+1] = A_d x[k] + B_0(D)Kx[k] + B_1(D)Kx[k-1]. \tag{13}$$

In (13), we assume that $u[-1] = 0$ for $k = 0$. Next, we define new system states $z[k] = \begin{bmatrix} x[k] & x[k-1] \end{bmatrix}'$ and we get,

$$z[k+1] = A_{cl}(h, D_i)z[k], \tag{14}$$

where

$$A_{cl}(h, D_i) = \begin{bmatrix} 0 & \Lambda \\ B_1(D)K & A_d + B_0(D)K \end{bmatrix}, \tag{15}$$

where Λ is the unity matrix. Stability of the overall closed-loop system is governed by the properties of $A_{cl}(h, D)$ and for stability, the absolute value of maximum eigenvalue of $A_{cl}(h, D)$ should be less than unity, i.e.,

$$|\lambda_{max}(A_{cl}(h, D))| < 1. \tag{16}$$

The closed-loop system might become unstable when D is long and fails to meet (16).

II-C. Quality of Control

In an ideal implementation, the execution delay of a control application is $D = 0$. An actual implementation under resource constraints results in $D > 0$ which causes deterioration in the performance of a control loop. As a QoC measure, we consider *stability margin* of a control loop, i.e.,

$$J = 1 - |\lambda_{max}(A_{cl}(h, D))|. \tag{17}$$

Clearly, the stability margin quantifies how far is the feedback loop from being *unstable*. As a QoC metric, we consider the degradation in control performance due to implementation irregularities (such as $D > 0$) from their performance with ideal implementation (where $D = 0$). The performance with ideal implementation is defined as *nominal* performance J_0, where

$$J_0 = 1 - |\lambda_{max}(A_{cl}(h, 0))|. \tag{18}$$

With $D > 0$ in an actual implementation, we have

$$J(D) = 1 - |\lambda_{max}(A_{cl}(h, D))|. \tag{19}$$

To this end, we define the *QoC gradient* $\mathcal{P}(D)$ as

$$\mathcal{P}(D) = \frac{J_0 - J(D)}{J_0}. \tag{20}$$

Clearly, $\mathcal{P}(D)$ captures the rate of control performance degradation in relation to non-zero execution delay.

III. SCHEDULE SYNTHESIS SCHEME

In this section we first describe the setting under consideration, followed by the basic scheme of our proposed scheduling algorithm. Next, we explain the problem complexity and outline the details of our algorithm followed by a discussion on the algorithm complexity. Finally, we show the applicability of our results using an illustrative example.

III-A. System Description

We consider a system that consists of a task set τ which is mapped on a resource r with total available service β_r. Let $\tau_{rt} \subseteq \tau$ denote the set of time-critical real-time tasks, and $\tau_c \subseteq \tau$ be the set of safety-critical control tasks. Assume that all tasks are mapped and executed on resource r sharing service β_r. Further, the set of indices is denoted by $i \in I$, and $I_{rt} \subseteq I$ is the set of indices related to τ_{rt}, similarly $I_c \subseteq I$ denotes the set of indices related to τ_c. Let task $T_i \in \tau$, be characterized by the tuple $\{\alpha_i, e_i, d_i, \pi_i\}$ where

- $\alpha_i = (\alpha_i^u, \alpha_i^l)$ denotes a pair of arrival curves that triggers the activation of T_i

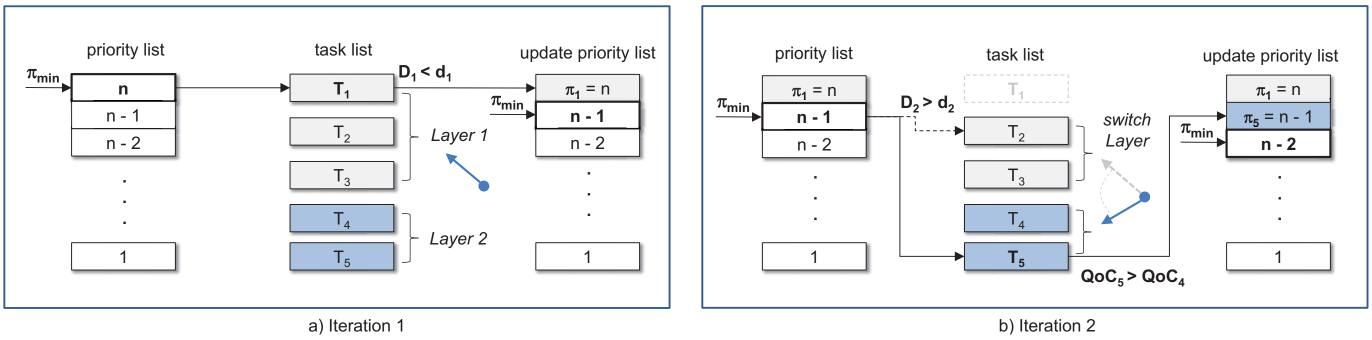

Fig. 4. Multi-layered scheduling scheme

- $e_i = (e_i^u, e_i^l)$ specifies the maximum and the minimum execution demand of T_i
- d_i represents the deadline of all real-time tasks $T_i \in \tau_{rt}$. For $T_i \in \tau_c$, i.e., control tasks, the deadline can be understood as the maximum delay bound at which the control system is still stable
- $\pi_i \in \{1, 2, ..., n\}$ denotes the priority of T_i on resource r where $\pi_i = 1$ represents the highest priority, $\pi_i = n$ the lowest priority, respectively

Further, the real-time task set τ_{rt} is schedulable iff *all* the tasks $T_i \in \tau_{rt}$ meet their deadlines, i.e., $D_i < d_i$. On the other hand there exists a maximum execution delay that a control task can tolerate until it gets unstable. However, in contrast to the real-time tasks, it is not sufficient to just meet this constraint but it is also necessary to optimize QoC. The control task set τ_c is schedulable iff *all* the tasks $T_i \in \tau_c$ meet their stability constraints (16). On top of stability the goal is to maximize the overall QoC, $J^* = \sum_{i \in I_c} J_i$.

Further, we assume a FPPS policy which is supported by many operating systems in various industrial domains such as OSEK/VDX [13], a widely-used standard in the automobile industry.

III-B. Multi-layered Scheduling Scheme

Before we present the details of our proposed schedule synthesis algorithm we first outline the basic scheme using the example in Fig. 4. The overall goal is to come up with a priority assignment strategy such that all real-time tasks $T_i \in \tau_{rt}$ meet their firm deadlines d_i, and at the same time, overall QoC is optimized for the control tasks $T_i \in \tau_c$. For this we present a multi-layered scheduling scheme where tasks of similar criticality are assigned a certain layer in the scheduling engine. This allows to implement appropriate scheduling strategies to each type of criticality in an integrated scheduling framework. In the example, we have assigned the real-time task set τ_{rt} the top layer, and the control task set τ_c the bottom layer. Let $T_1, T_2, T_3 \in \tau_{rt}$ be the set of real-time tasks and $T_4, T_5 \in \tau_c$ be the set of control tasks as illustrated in the figure. Further, let $n = |\tau|$ denote the total number of tasks in the system. In the example, $n = 5$, and hence, we have five priorities to be assigned. The available priorities $\{1, ..., n\}$ are indicated by a *priority list*, where the minimum available priority is denoted by π_{min}, i.e., the lowest priority which has not been assigned to any task yet. Similarly, the *task list* represents all the tasks $T_i \in \tau$ that have not been assigned any priorities. Note that the task list basically consists of two lists representing the different layers:

- **Layer 1:** Contains the set of real-time tasks τ_{rt} which is sorted according to deadlines d_i in descending order. Tasks $T_i \in \tau_{rt}$ are assigned priorities with the goal to guarantee the deadlines, i.e., $D_i \leq d_i$.
- **Layer 2:** Contains the set of unsorted control tasks τ_c. Tasks $T_i \in \tau_c$ are assigned priorities with the goal to guarantee stability as per (16) and to maximize overall QoC (J^*) according to the *QoC gradient* (see (20)).

Let us look at the example in Fig. 4a). We start assigning the minimum available system priority $\pi_{min} = n$ to the first task $T_1 \in \tau_{rt}$ of *Layer 1*, i.e., $\pi_1 = \pi_{min}$, and check whether $D_1 \leq d_1$ holds. In the example $D_1 < d_1$, and hence, T_i meets its deadline if assigned π_{min}. Consequently, we update the priority and task list for the next iteration as illustrated in Fig. 4a). This requires, (i) fixing the priority assignment $\pi_1 = n$, (ii) decrementing the minimum available system priority $\pi_{min} = n - 1$ for the next iteration, and (iii) removing T_1 from the task list. We repeat this procedure for every task in *Layer 1* until the schedulability test fails. For instance, the worst-case delay experienced by T_2 exceeds the deadline $(D_2 > d_2)$ for $\pi_2 = \pi_{min}$. As T_2 is currently the real-time task with the longest deadline, also none of the other remaining real-time tasks will be able to meet its deadline while being assigned $\pi_i = \pi_{min}$. Hence, we need not to check the other real-time tasks for schedulablility, and we *switch* to *Layer 2* where we now pursue a QoC-oriented priority assignment strategy. In particular, we assign π_{min} to the control tasks $T_4, T_5 \in \tau_c$ for which overall QoC is optimized, e.g., T_5. Subsequently, the priority and task list is updated again, i.e., $\pi_5 = n - 1$ is locked, $\pi_{min} = n - 2$, and T_5 is removed from the task list. This procedure is repeated until all tasks $T_i \in \tau$ have been successfully assigned priorities.

III-C. Complexity of the Schedule Synthesis Problem

Clearly, assigning priorities to real-time tasks according to Deadline Monotonic (DM) can be done in polynomial time. However, a priority assignment for the control tasks requires more effort, in particular, since this normally requires to maximize the overall control performance in the system.

As stated before, the control performance of a control task is a function of its execution delay: the more execution delay, the worse the performance. However, some control applications are more sensitive to delay than others. Moreover, reducing the execution delay of a control task T_i by some amount Δ might result in a performance increase which is less than that of reducing a T_j's delay by the same amount Δ. As a consequence, finding a priority assignment that leads to the maximum possible control performance requires analyzing the behavior of all control tasks for all possible priority levels.

In general, if the execution delay of all control tasks is the minimum possible, the overall control performance will be the maximum possible. However, in a mixed-critically environment, since control tasks share service on a resource (not only among them but also with real-time tasks), the execution delay suffered by them will vary with the priority they are assigned to: the higher the priority assigned to a control task, the lower its execution delay. (Of course, in a mixed-critically environment, not all priority levels can be assigned to a control task, since some priorities will necessarily be used by real-time tasks to guarantee their deadlines.)

On the one hand, which control task is assigned a given priority π_i depends on the control task's performance behavior for the execution delay resulting of π_i. On the other hand, this also depends on the execution delays incurred by other control tasks because of not being assigned π_i but other priorities. In other words, the optimal priority assignment for control tasks (i.e., the one that maximizes the overall control performance) in a mixed-criticality environment requires analyzing all possible priority combinations. That is, if n is the number of all tasks in the system, finding the optimal priority assignment requires $\mathcal{O}(n!)$ time (i.e., factorial of n), which results in an exponential complexity on the number of tasks in the system.

An algorithm with exponential complexity is not suitable for large- or even mid-size problem settings. For this reason, in this paper, we propose a heuristic which allows for a near-optimum priority assignment as detailed below.

III-D. Schedule Synthesis Algorithm

In this section we present the detailed algorithm of our proposed schedule synthesis scheme illustrated in Alg. 1.

Before the actual algorithm starts, we first perform an initialization process which requires

- sorting the real-time task set τ_{rt} according to deadlines d_i in descending order (line 1)
- initialization of the minimum available system priority $\pi_{min} = n$ (line 2)
- initialization of all task priorities $\pi_i = 0,\ \forall i \in I$ (line 3). Tasks with priorities $\pi_i = 0$ indicate tasks in the task list whereas tasks with $\pi_i = \{1, ..., n\}$ indicate tasks which have already been successfully assigned feasible priorities, and hence do not belong to the task list

As long as there exist unassigned priorities (line 4) we iterate over all real-time tasks in the task list, i.e. $\{T_i \in \tau_{rt} \mid \pi_i = 0\}$ (line 5) while $T_i \in \tau_{rt}$ is assigned the actual minimum priority $\pi_i = \pi_{min}$ (line 6). Next, corresponding to π_i, we compute the worst-case delay D_i calling the function *computeDelay()* (line 7) using the RTC framework. This involves the following steps outlined in Alg. 2:

- Computation of the lower remaining service $\beta^l_{r,i}$ for T_i. This is achieved using the relation defined in (9). From the total available service β_r we deduct the execution demands e_j of all event streams α_j of the tasks $T_j \in \tau$ with higher priorities than T_i's, i.e., tasks for which $\pi_j < \pi_i$ (lines 1-5). Recall that $\pi_i = \pi_{min}$, and the priorities of all unscheduled tasks have been initialized with $\pi_i = 0$. In other words, all tasks T_j that have not been assigned any priorities, i.e., $\pi_j = 0$, are considered as tasks having higher priorities than T_i.
- Computation of the worst-case delay experienced by the event stream α_i (line 6) using $D_i = \mathsf{hdist}(\alpha^u_i, \beta^l_{r,i})$, i.e., $D_i = \sup\big\{ \inf\{\tau \geq 0 \mid \alpha^u_i(t) \leq \beta^l_{r,i}(t+\tau)\} \ \mid \ t \geq 0\big\}$

Next, we check if D_i respects the deadline and in case $D_i \leq d_i$ (line 8) we lock the priority assignment $\pi_i = p_{min}$

Algorithm 1 Priority assignment algorithm.

```
Require: $\tau, \beta_r$
 1: sort $\tau_{rt}$ according to deadlines
 2: $\pi_{min} = n$
 3: $\pi_i = 0,\ \forall i \in \{1, ...n\}$
 4: while $\pi_{min} > 0$ do
 5:   for all $\{i \in I_{rt} \mid \pi_i = 0\}$ do
 6:     $\pi_i = \pi_{min}$
 7:     $D_i$ = computeDelay()
 8:     if $(D_i \leq d_i)$ then
 9:       $T_i$ is assigned priority $\pi_i = \pi_{min}$
10:     else
11:       $T_i$ is assigned invalid priority $\pi_i = 0$
12:       if $\{\exists j \in I_c \mid \pi_j = 0\}$ then
13:         if (selectControlTask() $== stable$) then
14:           $\pi_{min} = \pi_{min} - 1$
15:         else
16:           return (not schedulable)
17:         end if
18:       else
19:         return (not schedulable)
20:       end if
21:     end if
22:     $\pi_{min} = \pi_{min} - 1$
23:   end for
24:   if $\{\forall i \in I_{rt} \mid \pi_i \neq 0\}$ then
25:     if (selectControlTask() $== stable$) then
26:       $\pi_{min} = \pi_{min} - 1$
27:     else
28:       return (not schedulable)
29:     end if
30:   end if
31: end while
32: return (schedulable)
```

Algorithm 2 *computeDelay()*

```
1: $\beta^{l'} = \beta_r^{l'}$
2: for all $\{j \in I \mid \pi_j < \pi_i\}$ do
3:     $\beta^{l'} = \left(\beta^{l'} - \alpha_j^u\right) \overline{\otimes}\ 0$
4: end for
5: $\beta_{r,i}^l = \beta^{l'}$
6: $D_i = \mathsf{hdist}(\alpha_i^u, \beta_{r,i}^l)$
7: return $(D_i)$
```

Algorithm 3 *selectControlTask()*

```
1: $stable = false$
2: for all $\{i \in I_c \mid \pi_i = 0\}$ do
3:     $\pi_i = \pi_{min}$
4:     $D_i = computeDelay()$
5:     if $(|\lambda_{max}(A_{cl}(h_i, D_i))| < 1)$ then
6:         $stable = true$
7:         $\mathcal{P}_i = QoCGradient()$
8:     end if
9: end for
10: if $(stable == true)$ then
11:     for all $\{i \in I_c \mid (|\lambda_{max}(A_{cl}(h_i, D_i))| < 1)\}$ do
12:         if $(\mathcal{P}_i = \min_i \mathcal{P}_i)$ then
13:             $\pi_i = \pi_{min}$
14:         else
15:             $\pi_i = 0$
16:         end if
17:     end for
18:     return (stable)
19: else
20:     return (unstable)
21: end if
```

(line 9). In case the deadline is violated, we reset the priority $\pi_i = 0$ (line 11) because T_i requires a higher priority to guarantee its deadline. Further, in case there exists at least one unscheduled control task (line 12) we call the function *selectControlTask()* (line 13) which is responsible for the priority assignment to the control tasks. In case no control task is available the task set τ is not schedulable (line 19). Function *selectControlTask()* involves the following steps outlined in Alg. 3:

- Computation of worst-case delay D_i (line 4) (see Alg. 2), check for stability according to (16) (line 5), and computation of the *QoC gradient* $\mathcal{P}_i$ (line 7).
- Check if at least one control task is stable (line 10) and return *unstable* otherwise. Note that if Alg. 3 returns *unstable*, Alg. 1 declares the task set τ as *unschedulable* (lines 16 and 28) and the algorithm will stop. Next, the task with minimum $\mathcal{P}_i$ (lines 11-17) is assigned π_{min}. Note that if there are two tasks with minimum QoC, then π_{min} is assigned to one of them arbitrarily.

In case a feasible priority assignment was found for a real-time task (line 9) or a control task (line 13), π_{min} is decremented for the next iteration (line 14, line 22 of Alg. 1, respectively). Note that lines 24 - 30 again represent the priority assignment algorithm for the control tasks in case all real-time tasks already have been scheduled and there are only control tasks left.

III-E. Complexity Analysis

In Section III-C we analyzed the complexity of finding and optimal schedule for mixed control/real-time tasks settings. In this section we are concerned with analyzing the complexity of our proposed schedule synthesis algorithm.

As discussed previously, it is meaningful to assign priorities to real-time tasks in the system according to DM. For this purpose, the set of real-time tasks τ_{rt} needs to be sorted in order of decreasing deadlines (see Line 1 in Alg. 1). Recall that our schedule synthesis algorithm starts assigning priorities from the lowest to the highest. This way, if real-time tasks are schedulable at lower priority levels, the algorithm can use higher priority levels to accommodate control tasks and, hence, improve QoC. Such sorting can be performed in $\mathcal{O}(|\tau_{rt}| \log |\tau_{rt}|)$ time where $|\tau_{rt}|$ represents the number of elements in τ_{rt}. Further, our proposed algorithm outlined in Alg. 1 has to assign n different priorities, i.e., for each priority level, there will be either one real-time or one control task – multiple tasks per priority level are not allowed in our setting. For each priority level, there is one iteration of the while-loop (line 4). Hence, this loop is executed n times starting by $\pi_{min} = n$ (see Line 2) until $\pi_{min} = 0$ is reached (see Line 4).
In each iteration of the while-loop, the algorithm tries to accommodate a real-time task (from the sorted task set τ_{rt}) in the current priority level given by π_{min} (see for-loop, Line 5). For this, the delay incurred by the real-time task at the current π_{min} needs to be computed (see *computeDelay()*, Line 7). The function *computeDelay()* (shown in Alg. 2) computes the *minimum* service $\beta_{r,i}^l$ which results from subtracting the arrival curves α_j^u of higher-priority events to the resource's service curve. It has been proven that if a deadline is missed by a given schedule, this always happens within the so-called *busy period*, i.e., within the time interval in which the resource is busy without idle time [11]. As a consequence, the computation of $\beta_{r,i}^l$ can be limited to the busy period on the resource. Hence, *computeDelay()*'s complexity is pseudo-polynomial in the form $\mathcal{O}(n \times Z)$, where n is the total number of tasks in the system and Z is the length of the *busy period* on the shared resource [2].

The function *selectControlTask()* (shown in Alg. 3) calls *computeDelay()*. Since *selectControlTask()* iterates over the set of control tasks, its complexity is $\mathcal{O}(|\tau_c| \times n \times Z) < \mathcal{O}(n^2 \times Z)$, where $|\tau_c|$ denotes the number of tasks in τ_c. Note that the function *QoCGradient()* in Alg. 3 has constant complexity and hence does not influence *selectControlTask()*'s complexity.

The for-loop's body in Alg. 1 can be executed at most n times (once for each priority level). (Note that, if a real-time task is not be schedulable for a given priority level π_{min}, it will require another iteration of the for-loop. However, the number of iterations in the for-loop is upper-bounded

Table I. Task set specification

T_i	$\alpha := (p, j, d)$	execution demand e_i	deadline d_i
$T_1 \in \tau_{rt}$	(50,7,1)	(1.5 0.2)	50
$T_2 \in \tau_{rt}$	(30,4,1)	(1.3 0.2)	30
$T_3 \in \tau_{rt}$	(20,10,1)	(0.6 0.2)	20
$T_4 \in \tau_{rt}$	(20,25,1)	(0.8 0.3)	15
$T_5 \in \tau_{rt}$	(10,55,1)	(1.0 0.5)	10
$T_6 \in \tau_{rt}$	(10,19,1)	(1.2 0.1)	10
$T_7 \in \tau_c$	(30,0,1)	(1.4 0.2)	(14)
$T_8 \in \tau_c$	(40,0,1)	(1.2 0.3)	(32)
$T_9 \in \tau_c$	(40,0,1)	(0.7 0.1)	(27)
$T_{10} \in \tau_c$	(30,0,1)	(1.0 0.2)	(21)

Table II. Results for DM and QoC-oriented scheduling

Task T_i	*Deadline Monotonic:* (π_i)	*QoC Gradient:* $\mathcal{P} : (\pi_i)$
T_1	$\pi_1 = 10$	$\pi_1 = 10$
T_2	$\pi_2 = 8$	$\pi_2 = 9$
T_3	$\pi_3 = 5$	$\pi_3 = 7$
T_4	$\pi_4 = 4$	$\pi_4 = 4$
T_5	$\pi_5 = 2$	$\pi_5 = 3$
T_6	$\pi_6 = 1$	$\pi_6 = 2$
T_7	$\pi_7 = 3$	$\pi_7 = 1$
T_8	$\pi_8 = 9$	$\pi_8 = 8$
T_9	$\pi_9 = 7$	$\pi_9 = 6$
T_{10}	$\pi_{10} = 6$	$\pi_{10} = 5$
$\mathcal{P}^*$	2.7297	1.427235
J^*	0.32799	0.875653

by n.) The complexity of executing the for-loop in Alg. 1 is given by $\mathcal{O}\left(n \times (n \times Z + n^2 \times Z\right)$ which can be expressed as $\mathcal{O}\left(n^3 \times Z\right)$, i.e., also pseudo-polynomial as a result of the complexity of *computeDelay()*. Finally, since the while-loop's body is also executed n times, the overall complexity of our proposed algorithm is pseudo-polynomial in the form $\mathcal{O}\left(n^4 \times Z\right)$. This is a useful result since, as discussed before, the problem has exponential complexity.

III-F. Illustrative Examples

Let us consider the task set τ that consists of real-time tasks τ_{rt} and control tasks τ_c with the specification of Tab. I. The deadlines (in brackets) of the control tasks T_7, T_8, T_9, T_{10} represent the maximum delay at which the inequality in (16) is fulfilled. Further, we consider a resource r with a total initial service β_r. We compare the results of our proposed approach with classical DM scheduling. Tab. II shows the priorities π_i assigned to each task T_i and the total QoC defined as $J^* = \sum_i J_i$ and the overall *QoC gradient* $\mathcal{P}^* = \sum_i \mathcal{P}_i$, for all $i \in I_c$. It can be observed that the overall QoC for classical deadline monotonic scheduling is significantly worse compared to the QoC-oriented approach.

IV. CONCLUDING REMARKS

This paper presents a QoC-oriented schedule synthesis algorithm for mixed-criticality systems. The proposed approach integrates RTC-based performance analysis techniques in a QoC-oriented multi-layered scheduling framework where task sets of different criticality are scheduled at different layers. We compare our results to classical DM scheduling, and we show that our approach significantly improves overall QoC while guaranteeing schedulability. Future work envisages to extend the scheduling framework by adding additional layers, e.g., for soft real-time tasks, and integrating further QoC metrics for schedule synthesis.

V. REFERENCES

[1] S. Baruah, Haohan Li, and L. Stougie. Towards the design of certifiable mixed-criticality systems. In *RTAS*, 2010.

[2] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS*, 1990.

[3] S.K. Baruah, A. Burns, and R.I. Davis. Response-time analysis for mixed criticality systems. In *RTSS*, 2011.

[4] A. Y. Bhave and B. H. Krogh. Performance bounds on state-feedback controller with network delay. In *CDC*, 2008.

[5] J.-Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. LNCS 2050, 2001.

[6] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *DATE*, 2003.

[7] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the scheduling of mixed-criticality real-time task sets. In *RTSS*, 2009.

[8] S. Islam, R. Lindstrom, and N. Suri. Dependability driven integration of mixed criticality SW components. In *ISORC*, 2006.

[9] K. Lakshmanan, D. de Niz, and R. Rajkumar. Mixed-criticality task synchronization in zero-slack scheduling. In *RTAS*, 2011.

[10] J. Y Leung. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *Performance Evaluation*, 1982.

[11] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in hard real-time environments. In *Journal of the Association for Computing Machinery*.

[12] A. Mayr, R. Ploesch, and M Saft. Towards an operational safety standard for software: Modelling IEC61508 part 3. In *ECBS*, 2011.

[13] OSEK OS specification v2.2.3. www.osek-vdx.org.

[14] R. Palin, D. Ward, I. Habli, and R. Rivett. ISO26262 safety cases: Compliance and assurance. In *System Safety*, 2011.

[15] A. Quagli, D. Fontanelli, L. Greco, L. Palopoli, and A. Bicchi. Designing real-time embedded controllers using the anytime computing paradigm. In *ETFA*, 2009.

[16] J. Rushby. New challenges in certification for aircraft software. In *EMSOFT*, 2011.

[17] S. Samii, A. Cervin, P. Eles, and Z. Peng. Integrated scheduling and synthesis of control applications on distributed embedded systems. In *DATE*, 2009.

[18] S. Samii, P. Eles, Z. Peng, P. Tabuada, and A. Cervin. Dynamic scheduling and control-quality optimization of self-triggered control applications. In *RTSS*, 2010.