Analyzing Real-Time Scheduling of Cyber-Physical Resilience*

Björn Andersson[1], Dionisio de Niz[1], and Sagar Chaki[2]

1 Carnegie Mellon University
2 Mentor Graphics

Abstract

Cyber-Physical Systems (CPS) involve software executing on a computer that interacts with its physical environment. Common steps in the design and analysis of such systems are: model the physical environment, develop software to interact with this physical environment, specify timing requirements of software, configure the software (e.g., assign priorities), and then analyze the timing requirements for a given configuration. This approach works but tends to have low resilience to disruption. With the pervasive use of CPS, there is an increasing need for developing timing analysis methods that achieve increased resilience by modeling the linkage between the execution of software and the physical environment. In this paper, we present a new model that describes the current state of the physical environment in terms of how tolerant it is to disruption of the software system; we call this model *Cyber-Physical Resilience* (CPR). We present an exact schedulability test for this model and implement a tool that performs this schedulability test. Through evaluation of randomly-generated tasksets, we find that (i) for tasksets with at most five tasks, for all tasksets in our evaluation, our new schedulability test never took longer than 15h, (ii) in most cases, our new schedulability test finishes much faster (seconds/minutes), and (iii) thanks to our CPR model, our new schedulability test makes it possible to guarantee schedulability on a single processor even for tasksets with utilization 400%. We also find that our schedulability test can successfully analyze a model of a multi-UAV system from [1].

Digital Object Identifier 10.4230/LIPIcs.ECRTS.2018.YY

1 Introduction

Cyber-Physical Systems (CPS) involve software executing on a computer that interacts with its physical environment. Common steps [21] in the design and analysis of such systems are: model the physical environment, develop software to interact with this physical environment, specify timing requirements of software, configure the software (e.g., assign priorities), and then analyze the timing requirements for a given configuration. This approach works but tends to have low resilience to disruption. With the pervasive use of CPS, there is an increasing need for developing timing analysis methods that achieve increased resilience by modeling the linkage between the execution of software and the physical environment.

Although several works on real-time scheduling that aims to improve control performance exists, results on resilience of real-time scheduling are scarce—see Related work section of this paper. From the perspective of resilience, the ideas in [17] are particularly interesting. It introduces a collection of sets of states of the physical environment where a set is associated with an index (an integer) that indicates how many consecutive control actions that can be disrupted without jeopardizing safety. When a controller executes successfully, the possible successor states of the physical environment is in a set of states where this set is associated

* The last author did the work when he was with Carnegie Mellon University.

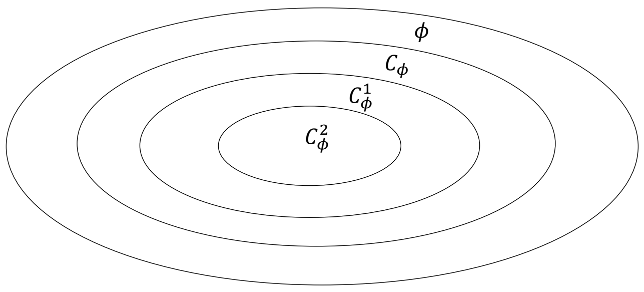


Figure 1 Increasing Enforceable Regions

with an index that is one *higher* (i.e., makes the plant *more* resilient). On the other hand, if a controller does not execute (e.g., because of a denial-of-service attack), the possible successor states of the physical environment is in a set of states where this set is associated with an index that is one *lower*. If the physical environment is in a set with index zero and the controller does not execute, then safety is violated. Despite the appeal of the model in [17] in terms of resilience, it had the drawback that it assumed a single controller—no contention for the processor and hence no scheduling. Specifically, [17] neither presented any task model nor any schedulability test. Therefore, we believe it is desirable to develop a scheduling theory inspired by the resilience model of [17].

In this paper, we present a new model that describes the current state of the physical environment in terms of how tolerant it is to disruption of the software system. The model is inspired by the ideas in [17] but we focus on real-time tasks, enforcement, interaction with physical environment, and schedulability analysis. Our model describes the relationship between the timing behavior and the physical resilience of the system; hence we name it the *Cyber-Physical Resilience* (CPR) model. For this model, we present an exact schedulability analysis. The main idea of our schedulability test is as follows (i) identify necessary conditions for a failure, (ii) formulate a Satisfiability Modulo Theories (SMT) instance such that if the necessary condition of failure is true then the SMT instance is feasible, (iii) by taking the contrapositive of the previous condition; obtain that if the SMT instance is unsatisiable then the taskset is schedulable (this is our new schedulability test), (iv) prove that our new schedulability test is exact. We also implement a tool that performs this schedulability test. Through evaluation of randomly-generated tasksets, we find that (i) for taskset with at most five tasks, for all tasksets in our evaluation, our new schedulability test never took longer than 15h, (ii) in most cases, our new schedulability test finishes much faster (seconds/minutes), and (iii) thanks to our CPR model, our new schedulability test makes it possible to guarantee schedulability on a single processor even for tasksets with utilization 400%. We also find that our schedulability test can successfully analyze a model of a multi-UAV system from [1].

We consider this research to be significant because it can be seen as a generalization of classic exact schedulability analysis of fixed-priority preemptive scheduling to cyber-physical systems. Previous work has ignored the physical dynamics or had explicit models of the physical dynamics; we, instead, use an abstraction of the dynamics of the physical world.

The remainder of this paper is structured as follows. Section 2 gives a background on the model in [17] and our observations. Section 3 formally presents our system model. Section 4 presents our new schedulability analysis. Section 5 presents performance evaluation. Section 6 presents related work. Section 7 concludes the paper.

2 Physical Interaction

Figure 1 shows notions that we will discuss. Consider a plant. Let ϕ be a correctness property of the plant (e.g., a UAV stays within a safe geographic region). We represent ϕ as a set of states of the plant where this correctness property is true. There is also a computer system that periodically senses and performs actuation actions on the plant. For a given pair of state and action there is a set of possible successor states.

Note that the state refers to the state of the plant—not the state of the computer system. Let C_ϕ be a subset of ϕ such that for each state in C_ϕ there is an action such that the set of possible successor states is a subset of C_ϕ.

Let a *null* action be an action such that the software does nothing. The interpretation of a null action is application specific and in particular, it depends on the actuator. For example, an actuator may be designed so that if it receives no new command, then it re-applies the command produced by the software in the previous period. Alternatively, an actuator may have an internal timeout so that if it has not received a new command from the software within a given timeout, then it actuates a specific command. For example, if a motor has not received any new command within a given timeout, then it may disengage. Let C^k_ϕ (for $k \geq 1$) denote the set of states such that for each state in C^k_ϕ, it holds that if k null actions are taken, then each possible successor state is in C_ϕ. Thus, both C^2_ϕ and C^1_ϕ are sets of safe states but C^2_ϕ is more resilient in the sense that it can tolerate more null actions without the plant reaching an unsafe state.

Lucia et al. [17] studied control of such a system where the computer can suffer from a Denial-of-Service attack, and it is desired to ensure that the system is still in a safe state after that. We will now describe how to use this idea to create a model for the real-time scheduling of the resilience of this type of systems. We will describe a system where the software consists of a set of tasks where each task generates a sequence of jobs. We assume that tasks operate on different plants, that is, task τ_i is associated with a plant i with the correctness property ϕ_i. A counter is associated with each task; if the counter associated with task τ_i is equal to k, then it means that plant i is in a state in $C^k_{\phi_i}$. A job can be skipped and then it outputs a null action. If a job is skipped, the counter of the corresponding task is decremented. If a job is not skipped it can happen that the job finishes with some margin before its deadline issuing a normal controller action; then the counter is incremented. If the job is not skipped and the job does not finish with some margin before its deadline, then within some margin before the deadline, the job is killed and an enforcer arrives; if the enforcer finishes before the deadline issuing an enforcer action, then the counter is unchanged. If the enforcer has not yet finished at its deadline, then the enforcer is killed (and a null action is output) and the counter is decremented.

We assume that each task is assigned a priority and fixed-priority preemptive scheduling is used. We assume that whether a job is skipped cannot be controlled by the scheduler but we have rules that bound this behavior. There are different reasons for a job being skipped. One reason is that the software system intentionally inserts a null action. For example, skipping one job may temporarily free up processing time that can be used by other tasks and this may give higher QoS. Another reason is a mixed-criticality context; a low-criticality task may choose to skip so that a high-criticality task (e.g., flight control) will meet its deadline.

We are interested in having very few assumptions on knowledge of execution times while still being able to provide pre-run-time guarantees. For this purpose, we can stipulate that if a job is of the type respectC (intuitively respect execution time), then the execution time is at most a given parameter; if the job is of the type not-respectC then execution time may be greater than this parameter. Clearly, if all jobs are not-respectC, then it is impossible to provide pre-run-time guarantees. Therefore, we also stipulate bounds on how many jobs can be not-respectC.

Typically, it is desired not only to maintain the plant in a safe state but also to achieve other objectives (for example, a UAV should stay within a geographical area but it should also follow waypoints). Hence, there are cases (where the plant is not close to the border of the set of safe states), when a successful action (a job finishing with some margin before its deadline) should not increment the counter of the task. We can model this by introducing a parameter per task and when the counter of a task has reached this parameter, then the counter is not permitted to be incremented further.

Our goal is to (i) formulate a task model based on the above ideas, (ii) present an exact schedulability test for this task model, and (iii) evaluate the new schedulability test.

3 System Model

Subsection 3.1 states notations that we will use. Subsection 3.2 states taskset parameters. Subsection 3.3 presents run-time behavior; it explains the meaning of taskset parameters. Subsection 3.4 defines the notion of schedulable, and based on that it defines the notion *schedulability test*.

3.1 Notation

Throughout this article, we use the following notation and abbreviations. "with respect to" is written as wrt. "left-hand side" is written as lhs. "right-hand side" is written as rhs. $\{x|f(x)\}$ denotes the set where an element x is in the set if and only if $f(x)$ is true. $\langle a,b\rangle$ is a tuple with two elements a and b. $[a,b]$ is an interval of real numbers (e.g., a time interval). $\{a..b\}$ is the set of integers $\geq a$ and $\leq b$.

We use dot to mean it holds that; for example, when we write "$\forall x\ Q(x).\ P(x)$" we mean forall x such that $Q(x)$ is true, it holds that $P(x)$ is true. In some cases, when Q is a set, we write "$\forall x \in Q.\ P(x)$" we mean forall x such that x is in the set Q, it holds that $P(x)$ is true. We will frequently use the above notations on tuples, for example when we write "$\forall\langle j,q\rangle$ $Q(j,q).\ P(j,q)$" we mean forall tuples $\langle j,q\rangle$ such that $Q(j,q)$ is true, it holds that $P(j,q)$ is true. We will assume that logical conjunction can be performed over a set of variables; if the set is empty, then the result is true. Ditto for logical disjunction.

In figures that show schedules, an arrow pointing upwards indicates the arrival of a job, an arrow pointing downwards indicates the absolute deadline of a job, and a solid vertical line will be used to indicate a time when the run-time system release an enforcement execution (if needed) for a job.

When we say increment without specifying the amount, it is assumed to mean increment by one. Ditto for decrement.

3.2 Static parameters

Table 1 shows an example of a system in our model. We consider a taskset τ and a single processor. Each task $\tau_i \in \tau$ is characterized by prio_i, T_i, D_i, Z_i, C_i, E_i, MAXCOUNT_i, and RC_i such that $(T_i \geq D_i \geq Z_i \geq C_i \geq 0) \wedge (D_i - Z_i \geq E_i \geq 0) \wedge (\text{MAXCOUNT}_i \in \mathbb{N}_{\geq 1}) \wedge (\text{RC}_i \in \mathbb{N}_{\geq 1})$. The interpretation is as follows. A task τ_i is assigned priority prio_i. A task τ_i generates a sequence of jobs where two consecutive jobs of τ_i have arrival times

$|\tau| = 2$

$\text{prio}_1 = 2$	$T_1 = 1.0$	$D_1 = 0.8$	$Z_1 = 0.64$	$C_1 = 0.50$	$E_1 = 0.10$	$\text{MAXCOUNT}_1 = 4$	$\text{RC}_1 = 1$
$\text{prio}_2 = 1$	$T_2 = 2.2$	$D_2 = 1.7$	$Z_2 = 1.40$	$C_2 = 0.61$	$E_2 = 0.25$	$\text{MAXCOUNT}_2 = 1$	$\text{RC}_2 = 1$

Table 1 An example of a system in our model.

separated by at least T_i time units and each job of τ_i has relative deadline D_i. If a job of τ_i is a respectC job, then C_i is an upper bound on the execution time of this job performed in the time interval from its arrival until Z_i time units after its arrival. For a job of τ_i, regardless of whether it is a respectC job, it holds that E_i is an upper bound on the execution time of this job performed in the time interval from Z_i time units after its arrival until D_i time units after its arrival. MAXCOUNT_i is the largest value that the counter of task τ_i may take at run-time. RC_i specifies that we assume it cannot happen that RC_i consecutive jobs of τ_i are not-respectC. We assume that priorities of tasks are unique, that is, $((i \neq j) \wedge (\tau_i \in \tau) \wedge (\tau_j \in \tau)) \Rightarrow (\text{prio}_i \neq \text{prio}_j)$. For convenience, we let $\text{hp}(i)$ denote the set of indices of tasks with higher priority than task τ_i.

3.3 Run-time behavior

Figure 2 shows a schedule that the taskset specified by Table 1 can generate. $\tau_{i,q}$ denotes the q^{th} job of task τ_i. The priority of a $\tau_{i,q}$ is equal to prio_i. A job can be a skipped job or not; if it is skipped, then there is a time at which it gets skipped. (For example, a job may be skipped when it arrives because the operating system decided this. Alternatively, the application software may decide that a job should be skipped and this type of skip will happen after the job has arrived.) A job can be respectC job or not. We say that a job of task τ_i *Zexpires* at time Z_i plus the arrival time of the job. Analogously, we also say that a job of task τ_i *Dexpires* at time D_i plus the arrival time of the job. For a job $\tau_{i,q}$, the normal mode of the job is the time interval from its arrival until it Zexpires. For a job $\tau_{i,q}$, the enforcement mode of the job is the time interval from when it Zexpires until it Dexpires. A job performs normal execution in its normal mode. A job performs enforcement execution in its enforcement mode. The following rules (for evolution of counters, whether a job is eligible, and how a job is selected for execution) apply at run-time:

1. When the system starts, the counter of τ_i is initialized to some non-negative value. The choice in this value may be done non-deterministically when the system starts.
2. When a job arrives, if it is not skipped when it arrives, it becomes eligible.
3. When a job arrives, if it is skipped when it arrives, it is set to non-eligible.
4. If for $\tau_{i,q}$, the $\text{RC}_\text{i} - 1$ preceding jobs of are not-respectC job, then $\tau_{i,q}$ is a respectC job.
5. If for $\tau_{i,q}$, the job is in its normal mode and the job is skipped, then it finishes, it becomes non-eligible, and the counter of τ_i is decremented. (A job can only be skipped in its normal mode.)
6. If for $\tau_{i,q}$, the job is in its normal mode and it is not skipped and the job has performed C_i units of normal execution, then the job finishes (note that a job can finish even if it performs less) and it becomes not-eligible.
7. If for $\tau_{i,q}$, the job finishes in its normal mode and it is not skipped and the counter of the τ_i is at most $\text{MAXCOUNT}_i - 1$, then the counter of τ_i is incremented. (note that if the counter of τ_i equals MAXCOUNT_i, then the counter is unchanged; hence, the counter cannot exceed MAXCOUNT_i).
8. When a job Zexpires, it leaves its normal mode and enters its enforcement mode.

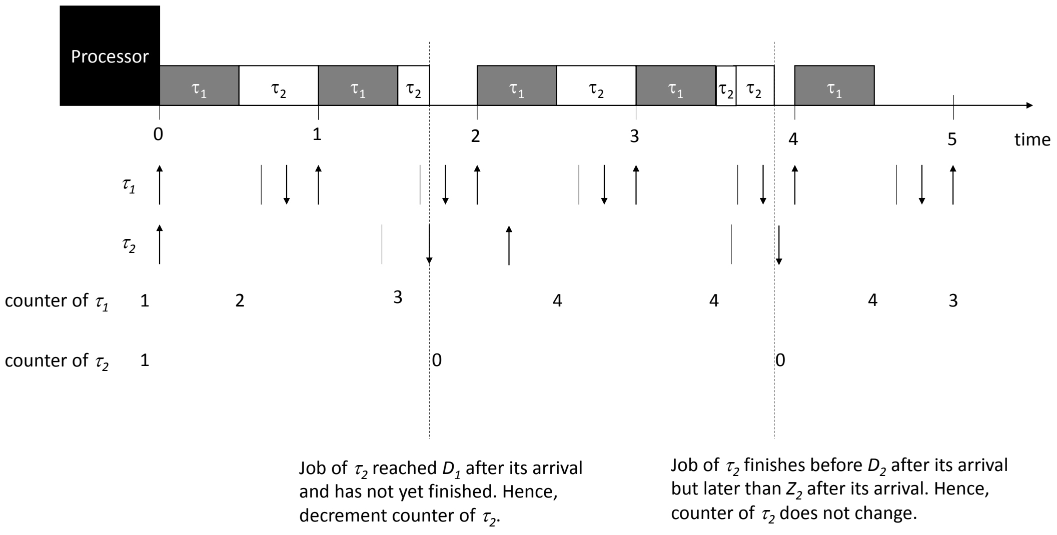

Processor
τ1
τ2
0
1
2
3
4
5
time
counter of τ1
counter of τ2
Job of τ2 reached D1 after its arrival and has not yet finished. Hence, decrement counter of τ2.
Job of τ2 finishes before D2 after its arrival but later than Z2 after its arrival. Hence, counter of τ2 does not change.

Figure 2 Illustration of a schedule that the system can generate. τ_1 has the highest priority; hence, a job of τ_1 executes immediately when it arrives. It can be seen that for each of the first five jobs of τ_1, the finishing time is at most Z_1 after its arrival. Because of this early finishing, for each of the first three jobs of τ_1, the counter of τ_1 is incremented. For the 4th and 5th job of τ_1, it holds that the job also has early finishing time but since the counter has already become 4 and $\text{MAXCOUNT}_1 = 4$, it follows that the counter is not incremented further. The 6th job of τ_1 arrives at time 5; this job is skipped and hence the counter of τ_1 is decremented. As a result, the counter of τ_1 becomes 3 at time 5. Note that in this schedule, it never happens that a job of τ_1 is eligible Z_1 time units after its arrival; hence, jobs of τ_1 never perform enforcement execution. τ_2 has lower priority; hence a job of τ_2 can only execute when a job of τ_1 is not executing. For the first job of τ_2, it holds that it has not yet finished Z_2 time units after its arrival; hence this job is killed and its enforcement execution is released (at time 1.4). Note that this job cannot execute immediately after time 1.4 because of the higher priority task. Therefore, this job has to wait until time 1.5 and then it performs enforcement execution during [1.5,1.7]. At time 1.7, the deadline of the 1st job of τ_2 expires and hence this job is killed and as a result, the counter of τ_2 is decremented. The 2nd job of τ_2 has similar behavior but since its enforcement execution experiences less interference, it finishes before the deadline and hence the counter of τ_2 is not changed.

9. If for $\tau_{i,q}$, the job is in its enforcement mode and the job has performed E_i units of enforcement execution, then the job finishes (note that a job can finish even if it performs less; also note that a job cannot be skipped in its enforcement mode) and it becomes not-eligible.

10. For $\tau_{i,q}$, if the job finishes in its enforcement mode, then the counter of τ_i does not change.

11. For $\tau_{i,q}$, when the job Dexpires, it finishes (here we say it is killed) and it becomes not-eligible and the counter of τ_i is decremented.

12. A job executes if and only if the job is eligible and there is no higher-priority job that is eligible at that time.

3.4 Schedulable

Informally, a system is schedulable if for all possible schedules that the system can generate, all deadlines are met. Therefore, we start by defining terminology for specifying the possible schedules.

Let R be an assignment, for each task, the number of jobs it generates and for each job, its arrival time and execution time in normal mode and in enforcement, whether it is a skip-job and the time when it becomes a skip job (only used if it is a skip job) and respectC. $\mathrm{nj}_i(R)$ denotes the $\underline{\mathrm{n}}$umber of $\underline{\mathrm{j}}$obs generated by τ_i for assignment R. Let $A_i(R)$ be the arrival time of $\tau_{i,q}$ for assignment R. Let $c_{i,q}(R)$ be the execution requirement of $\tau_{i,q}$ in normal mode for assignment R. Let $e_{i,q}(R)$ be the execution requirement of $\tau_{i,q}$ in enforcement mode for assignment R. Let $\mathrm{skip}_{i,q}(R)$ be a Boolean indicating whether $\tau_{i,q}$ is a skip-job for assignment R. Let $\mathrm{skipt}_{i,q}(R)$ be a real number indicating the time when $\tau_{i,q}$ becomes a skip-job (only used if it is a skip job) for assignment R. Let $\mathrm{respC}_{i,q}(R)$ be a Boolean indicating whether $\tau_{i,q}$ is respectC for assignment R. Let $\mathrm{leg}(R,\tau)$ mean $\underline{\mathrm{leg}}$al assignment. Formally:

$$\begin{aligned}\mathrm{leg}(R,\tau) = \Big(\forall\langle i,q\rangle\ (\tau_i\in\tau)\wedge(q\in\{2..\mathrm{nj}_i(R)\})\ A_{i,q}(R)-A_{i,q-1}(R)\geq T_i\Big)\wedge\\ \Big(\forall\langle i,q\rangle\ (\tau_i\in\tau)\wedge(q\in\{1..\mathrm{nj}_i(R)\})\ (c_{i,q}(R)\geq 0)\wedge(\mathrm{respC}_{i,q}(R)\Rightarrow(c_{i,q}(R)\leq C_i))\Big)\wedge\\ \Big(\forall\langle i,q\rangle\ (\tau_i\in\tau)\wedge(q\in\{1..\mathrm{nj}_i(R)\})\ e_{i,q}(R)\in[0,E_i]\Big)\wedge\\ \Big(\forall\langle i,q\rangle\ (\tau_i\in\tau)\wedge(q\in\{1..\mathrm{nj}_i(R)-(\mathrm{RC}_i-1)\})\ \vee_{q'\in\{q..q+\mathrm{RC}_i-1\}}\mathrm{respC}_{i,q'}(R)\Big)\end{aligned}$$

Let ca be a $\underline{\mathrm{c}}$ounter $\underline{\mathrm{a}}$ssignment; that is, it is an assignment for each task, for each time, the value of the counter of the task at that time. Let $\mathrm{legca}(\mathrm{ca},R,\mathrm{sc},\tau)$ mean that ca is $\underline{\mathrm{leg}}$al $\underline{\mathrm{c}}$ounter $\underline{\mathrm{a}}$ssignment related to the assignment R, schedule sc, τ; it means that the counter assignment respects the increment and decrement rules specifies in the Subsection 3.3.

We say that τ is *schedulable* if and only if for each assignment R, counter assignment ca, schedule sc s.t. R is legal wrt to τ, counter assignment ca is legal wrt τ and R, sc can be generated from R it holds that for each task $\tau_i\in\tau$, at all times, the counter of τ_i is non-negative. The intuition behind this definition of schedulable is (as mentioned in Section 2) that counters that are non-negative represent safe states; thus, if a system is schedulable, then it is safe.

A schedulability test is a function that takes τ and Π as input and outputs a Boolean. For an *exact* schedulability test, it holds that if and only if the schedulability test outputs true, then the system is schedulable.

4 New Schedulability Test

In this section, we present our new schedulability test for the CPR model. Our plan is as follows. First, if a taskset is unschedulable, then there is a schedule in which there is a failure. We reason (in Section 4.1) about this failure and obtain the existence of another schedule that the system can generate; for this schedule, we identify necessary conditions. Then, we represent those schedules with variables and constraints. Thus, if a taskset is unschedulable, then there exists a constraint satisfaction problem that is satisfiable (Section 4.2 presents this constraint satisfaction problem). Then, in Section **??** we show that we have obtained a necessary condition for an unschedulable taskset. We take the contrapositive of this necessary condition and this yields a sufficient condition for a taskset to be schedulable.

We then show (in Section 4.3) that this condition is also exact. It turns out that it is not obvious how to evaluate this condition. Hence, we develop (in Section 4.4) an algorithm to evaluate the condition and this yields our exact schedulability test for the CPR model.

4.1 Reasoning About Necessary Condition for Failure

Consider an unschedulable taskset τ. From the definition of schedulability, it follows that

> There is an assignment R, a counter assignment ca, a schedule sc, a task τ_{iD}, and an instant t_0 such that (i) R is legal wrt the taskset τ, (ii) ca is legal wrt R, sc, and τ, (iii) sc can be generated by R and τ, and (iv) just before t_0, the counter of τ_{iD} was 0 and at t_0, the counter of τ_{iD} becomes -1.

If there are multiple such t_0, then choose the earliest one. Hence, for each task, for each instant before t_0, it holds that at this instant the counter of this task is non-negative. We will present transformations such that after each transformation, the above three conditions (and the non-negativeness) are true but in addition, there are other constraints that are true as well. We will now perform a transformation as specified by the following steps:

T1 For tasks of lower priority than τ_{iD}, set the number of jobs to zero.
T2 Remove all jobs that arrive after t_0.
T3 For each task, set the counter of this task as a function of time to reflect the changes of steps T1 and T2.

Note that even after this change, it holds that at time t_0, the counter of τ_{iD} becomes -1. Recall that according to our system model, there are two reasons why the counter of a task is decremented: either because a job of this task is skipped or because the job finishes too late. The former cannot happen when the counter is zero (which is the case just before time t_0). Hence, at time t_0, the counter of τ_{iD} is decremented because a job of τ_{iD} finishes too late. We let qD denote the index of this job of task τ_{iD}. In addition, recall from our system model that if a job has not finished by its absolute deadline, then the job is killed at its absolute deadline; thus $\tau_{\mathrm{iD,qD}}$ is killed a time t_0. We will now perform a transformation as specified by the following steps:

T4 For all jobs except $\tau_{\mathrm{iD,qD}}$, if the job performs some enforcement execution at time t_0 or later, then reduce the enforcement execution time of the job; keep doing this until it performs no enforcement execution after t_0.
T5 For all jobs except $\tau_{\mathrm{iD,qD}}$, if the job performs some normal execution at time t_0 or later, then reduce the normal execution time of the job; keep doing this until it performs no normal execution after t_0.
T6 For each task, set the counter of this task as a function of time to reflect the changes of steps T4-T5.

Note that even after this change, it holds that at time t_0, the counter of τ_{iD} becomes -1. Hence, it holds that:

> There is an assignment R, a counter assignment ca, a schedule sc, a task τ_{iD}, and an instant t_0, and a positive integer qD such that (i) R is legal wrt the taskset τ, (ii) ca is legal wrt R, sc, and τ, (iii) sc can be generated by R and τ, (iv) just before t_0, the counter of τ_{iD} was 0 and at t_0, the counter of τ_{iD} becomes -1, (v) for each task, for each instant before t_0, it holds that at that time, the counter of the task is non-negative, (vi) at each instant less than t_0, for each task, it holds that the counter of this task at this instant is non-negative, (vii) for each task of lower priority than τ_{iD} it holds that, the task generates no jobs, (viii) no jobs arrive after t_0, (ix) the number

of jobs generated by τ_{iD} is qD, (x) the absolute deadline of $\tau_{\mathrm{iD,qD}}$ is t_0, (xi) $\tau_{\mathrm{iD,qD}}$ has not yet finished by time t_0, (xii) $\tau_{\mathrm{iD,qD}}$ is killed at time t_0, and (xiii) the processor is idle at all times strictly after t_0.

Let us consider two cases based on the initial value of the counter of task τ_{iD}.

Case 1: When the system starts, the value of the counter of task τ_{iD} is at least 1.
Since at time t_0, it holds that the counter of τ_{iD} changes from 0 to -1 and since the counter of τ_{iD} started with at least 1, it holds that there are at least two jobs of τ_{iD} for which the counter of τ_{iD} was decreased. Hence $\mathrm{qD} \geq 2$. Let us define qD' as the smallest number such that after the absolute deadline of $\tau_{\mathrm{iD,qD}'}$ until t_0, the counter of task τ_{iD} does not change. Formally,

$$\mathrm{qD}' = \min_{q \mid \textit{the counter of } \tau_{\mathrm{iD}} \textit{ does not change during } (A_{\mathrm{iD},q}(R)+D_{\mathrm{iD}},t_0)} q$$

Note that since there are at least two jobs of τ_{iD} for which the counter of τ_{iD} was decreased, it follows that qD' exists. We will now perform a transformation as specified by the following steps:

T7 Set $\tau_{\mathrm{iD,qD}'}$ to be skipped; set the time when this job is skipped to be the time when the job arrives.

T8 Set $\tau_{\mathrm{iD,qD}'}$ to respectC.

T9 Remove all jobs $\tau_{\mathrm{iD},q}$ such that $((q \neq \mathrm{qD}') \wedge (q \neq \mathrm{qD}))$.

T10 Set the counter of τ_{iD} as a function of time to reflect the changes of steps T7-T9.

About the transformation, we note the following:

A The job $\tau_{\mathrm{iD},1}$ after this transformation refers to the same job as $\tau_{\mathrm{iD,qD}'}$ before the transformation.

B The job $\tau_{\mathrm{iD},2}$ after this transformation refers to the same job as $\tau_{\mathrm{iD,qD}}$ before the transformation.

C After the transformation, at time t_0, the counter of τ_{iD} changes from 0 to -1.

D After the transformation, there are two jobs of τ_{iD}.

E After the transformation, each of the two jobs of τ_{iD} causes the counter of τ_{iD} to be decremented.

F After the transformation, when the system starts, the counter of τ_{iD} is 1 (this follows from C. and E.).

G After the transformation, $\tau_{\mathrm{iD},1}$ does not execute (this follows from A and T7).

H After the transformation, $\tau_{\mathrm{iD},1}$ is a respectC job. (this follows from A and T8).

I If $\mathrm{RC}_{\mathrm{iD}} = 1$, then before the transformation, each job of τ_{iD} is a respectC job. (follows from the fact that R is legal).

J If $\mathrm{RC}_{\mathrm{iD}} = 1$, then after the transformation, $\tau_{\mathrm{iD},2}$ is a respectC job. (follows from I and B).

We will now show that after this transformation, the assignment and schedule are still legal. Clearly, this transformation does not change the arrival times, execution times of tasks with index in hp(iD); also, for these tasks, the times when they execute do not change, their counters do not change, and their respectC do not change (and hence RC constraints are satisfied too). We will now show that this also holds for the task τ_{iD}. Clearly, the execution time and arrival time constraints for τ_{iD} are satisfied. We will now show that the RC constraint for τ_{iD} is satisfied after the change. Consider the case that $\mathrm{RC}_{\mathrm{iD}} = 1$. From H. and J., it follows that after the transformation, both jobs of τ_{iD} are respectC jobs; hence the RC constraint for τ_{iD} is satisfied. Consider the case that $\mathrm{RC}_{\mathrm{iD}} \geq 2$. In order to prove that the RC constraint of τ_{iD} is satisfied after the change, it suffices to show that for each sequence of $\mathrm{RC}_{\mathrm{iD}}$ jobs of τ_{iD}, it holds that at least one of these jobs is a respectC job. Note (from H) that $\tau_{\mathrm{iD},1}$ after the transformation is a respectC job and hence each such sequence has a respectC job. Thus, the RC constraint of τ_{iD} is satisfied after the transformation. Hence, it holds that:

> There is an assignment R, a counter assignment ca, a schedule sc, a task τ_{iD}, and an instant t_0, and a positive integer qD such that (i) R is legal wrt the taskset τ, (ii) ca is legal wrt R, sc, and τ, (iii) sc can be generated by R and τ, (iv) just before t_0, the counter of τ_{iD} was 0 and at t_0, the counter of τ_{iD} becomes -1, (v) for each task, for each instant before t_0, it holds that at that time, the counter of the task is non-negative, (vi) at each instant less than t_0, for each task, it holds that the counter of this task at this instant is non-negative, (vii) for each task of lower priority than τ_{iD} it holds that, the task generates no jobs, (viii) no jobs arrive after t_0, (ix) the number of jobs generated by τ_{iD} is 2, (x) the absolute deadline of $\tau_{\mathrm{iD},2}$ is t_0, (xi) $\tau_{\mathrm{iD},2}$ has not yet finished by time t_0, (xii) $\tau_{\mathrm{iD},2}$ is killed at time t_0, (xiii) the processor is idle at all times strictly after t_0, and (xiv) when the system starts, the counter of τ_{iD} is 1.

Let t_{-1} denote the earliest time instant such that at each instant during $[t_{-1}, t_0]$, the processor is busy. We will now discuss the arrival time of $\tau_{\mathrm{iD},1}$ and $\tau_{\mathrm{iD},2}$. Let us define Δ as $A_{\mathrm{iD},2}(R) - t_{-1}$. From the definition of t_{-1} and from the fact that we consider work-conserving scheduling, it follows that $\Delta \geq 0$. We will now consider two cases and through reasoning, we will show that after these cases, we end up with $A_{\mathrm{iD},2}(R) = t_{-1}$.

Case 1a: $\Delta = 0$
Using the knowledge of the case ($\Delta = 0$) and the definition of Δ yields that $A_{\mathrm{iD},2}(R) = t_{-1}$.

Case 1b: $\Delta > 0$
For this case, perform a transformation as specified by the following steps:

T11 Decrement $A_{\mathrm{iD},1}(R)$ by Δ.
T12 Decrement $A_{\mathrm{iD},2}(R)$ by Δ.
T13 Set t_0 to the absolute deadline of $\tau_{\mathrm{iD},2}$.
T14 Apply T4,T5,T6 so that there is no execution after the time given by the new value of t_0.
T15 Set the counter of τ_{iD} as a function of time to reflect the changes of steps T11-T14.

Given that both jobs of τ_{iD} have their arrival times decreased by the same amount and the minimum inter-arrival time was respected before the transformation, the minimum inter-arrival time is still respected after the transformation. Also, after the transformation, we obtain $A_{\mathrm{iD},2}(R) = t_{-1}$.

Hence, regardless of the case, we end up with $A_{\mathrm{iD},2}(R) = t_{-1}$. Also, note that we end up with that the absolute deadline of $\tau_{\mathrm{iD},2}$ equals t_0 (this can be seen as follows: in Case 1a, this was true initially and we did not change it; in Case 1b, this was true initially and then we changed it—with T12—and then T13 made sure it is true). We will now perform a transformation as specified by the following steps:

T16 Remove the job $\tau_{\mathrm{iD},1}$.
T17 Remove all jobs arriving before t_{-1}.
T18 Left shift-the schedule by t_{-1} time units.
T19 For each task, set the counter of this task as a function of time to reflect the changes of steps T16-T18.

Note that after this transformation, we obtain that τ_{iD} generates a single job that arrives at time 0; at that time, the counter of τ_{iD} is zero; and D_{iD} time units later, the deadline of the single job of τ_{iD} expires at time D_{iD}. Hence, it holds that:

> There is an assignment R, a counter assignment ca, a schedule sc, a task τ_{iD}, and an instant t_0, and a positive integer qD such that (i) R is legal wrt the taskset τ, (ii) ca is legal wrt R, sc, and τ, (iii) sc can be generated by R and τ, (iv) just before t_0, the counter of τ_{iD} was 0 and at t_0, the counter of τ_{iD} becomes -1, (v) for each task, for each instant before t_0, it holds that at that time, the counter of the task is non-negative, (vi) at each instant less than t_0, for each task, it holds that the counter of this task at this instant is non-negative, (vii) for each task of lower priority than τ_{iD} it holds that, the task generates no jobs, (viii) no jobs arrive after t_0, (ix) the number of jobs generated by τ_{iD} is 1, (x) the absolute deadline of $\tau_{\mathrm{iD},1}$ is D_{iD}, (xi) $\tau_{\mathrm{iD},1}$ has not yet finished by time D_{iD}, (xii) $\tau_{\mathrm{iD},1}$ is killed at time D_{iD}, (xiii) the processor is idle at all times strictly after D_{iD}, (xiv) when the system starts at time zero, the counter of τ_{iD} is 0, (xv) $\tau_{\mathrm{iD},1}$ arrives at time zero, and (xvi) no job arrives before time 0.

[End-of-Case-1]

Case 2: When the system starts, the value of the counter of task τ_{iD} is 0.
If qD ≥ 2, then remove all jobs of τ_{iD} except $\tau_{\mathrm{iD,qD}}$. Now we have a situation with a single job of τ_{iD}. Let t_{-1} denote the earliest time such that the processor is busy during $[t_{-1}, t_0]$. Then, remove all jobs that arrive before t_{-1}. And then left-shift the schedule by t_{-1}. This yields the same situation as in the end of Case 1. [End-of-Case-2]

It can be seen that regardless of the case, we obtain that the statement just before [End-of-Case-1] is true. Let us now discuss skiptime of a job of a task in hp(iD). Let us define a scheduling event as an event where at least one of the following happens (i) a job arrives, (ii) a job finishes, (iii) a job Zexpires, or (iv) a job Dexpires. We will show that for each job, we can set skiptime to a time of a scheduling event. Given a job, consider three case (i) the job does not execute at all, (ii) the job executes and there exists a time after skiptime when the job executes, and (iii) the job executes and for all times after skiptime, the job does not execute. For the first case, set the skiptime of the job to the arrival time of the job. For the third case, set skiptime of the job to the minimum of the finishing time and the time of the Zexpire. We will now discuss the second case. Since, at the time skiptime of the job, there is a time after skiptime of the job when the job executes, it follows that the job is not a skipjob. Since it is not a skipjob, we can set skiptime to any value and it does not impact the schedule. We set the skiptime of the job to its arrival time. Applying the above yields that for each job, skiptime of the job equals the arrival time of the job or skiptime of the job equals the minimum of the finishing time or the Zexpire time. Hence, for each job of a task in hp(iD), the skiptime is at a time of a scheduling event. Also, we set the skiptime of $\tau_{\mathrm{iD},1}$ to its deadline (this is possible because this job is not a skipjob so any assignment of skiptime will not impact the schedule).

We will now discuss the counters. For each $j \in$ hp(iD) do the following (i) throughout the schedule, find the smallest value of the counter for task τ_j and then (ii) throughout the schedule, subtract the counter of τ_j by the number computed in (i). Hence, we obtain that for each task with index $j \in$ hp(iD), there exists a time when the counter of the τ_j is zero and the counter is never lower than 0. Also, it is easy to see that for each task with index j, it holds that there are at most $\lceil D_{\mathrm{iD}}/T_j \rceil$ jobs of task τ_j. Hence, for each task with index j, during its schedule, it holds that at each instant, the counter of task τ_j is at most $\lceil D_{\mathrm{iD}}/T_j \rceil$.

Also, from the system model, for each task with index j, during its schedule, it holds that at each instant, the counter of task τ_j is at most $\mathrm{MAXCOUNT}_j$. Putting them together yields that for each task with index j, during its schedule, it holds that at each instant, the counter of task τ_j is at most $\min(\lceil D_{\mathrm{iD}}/T_j \rceil, \mathrm{MAXCOUNT}_j)$. Hence, it holds that:

There is an assignment R, a counter assignment ca*, a schedule* sc*, a task τ_{iD}, and an instant t_0, and a positive integer* qD *such that (i) R is legal wrt the taskset τ, (ii)* ca *is legal wrt R,* sc*, and τ, (iii)* sc *can be generated by R and τ, (iv) just before t_0, the counter of τ_{iD} was 0 and at t_0, the counter of τ_{iD} becomes -1, (v) for each task, for each instant before t_0, it holds that at that time, the counter of the task is non-negative, (vi) at each instant less than t_0, for each task, it holds that the counter of this task at this instant is non-negative, (vii) for each task of lower priority than τ_{iD} it holds that, the task generates no jobs, (viii) no jobs arrive after t_0, (ix) the number of jobs generated by τ_{iD} is 1, (x) the absolute deadline of $\tau_{\mathrm{iD},1}$ is D_{iD}, (xi) $\tau_{\mathrm{iD},1}$ has not yet finished by time D_{iD}, (xii) $\tau_{\mathrm{iD},1}$ is killed at time D_{iD}, (xiii) the processor is idle at all times strictly after D_{iD}, (xiv) when the system starts at time zero, the counter of τ_{iD} is 0, (xv) $\tau_{\mathrm{iD},1}$ arrives at time zero, (xvi) no job arrives before time 0, (xvii) for each task with index in* hp(iD)*, for each job of the task, the skiptime of the job a time of a scheduling event, (xviii) for the job $\tau_{\mathrm{iD},1}$, its skiptime equals its deadline, and (xix) for each task with index j, at each instant, the counter of the task at this instant is at most* $\min(\lceil D_{\mathrm{iD}}/T_j \rceil, \mathrm{MAXCOUNT}_j)$. (0)

If we could find a simple function such that this function takes parameters of the taskset as input and outputs a Boolean such that (0) implies that the function is true, then we can obtain a schedulability test by negating the function. One could imagine that such a function could be obtained by summing up all the computation by jobs of tasks in hp(iD) and adding the normal and enforcement execution of $\tau_{\mathrm{iD},1}$ and then compare with D_{iD}. Unfortunately, doing so is very complicated in our model. The reason is that in our model, the amount of execution of a job depends on when the job finishes (if the job finishes before it Zexpires, then it performs no enforcement execution). Therefore, we will, instead, express the schedule in (0) with variables and constraints so that if and only if (0) is true, then the constraints are satisfiable.

4.2 Representing Schedules

General idea. We will express the schedule in (0) with variables and constraints so that if and only if (0) is true, then the constraints are satisfiable. Recall that (0) considers a schedule in the time interval $[0, D_{\mathrm{iD}}]$ and no jobs arrive before time 0 and there is no execution after time D_{iD}. Hence, we only need to consider at most $\lceil D_{\mathrm{iD}}/T_j \rceil$ jobs of task τ_j. Recall also from (1) that there is a single job of τ_{iD}. In addition, recall that there are four types of scheduling events (arrival, finishing, Zexpiring, Dexpiring) that can require a context switch (here, we consider that the instant when a job's normal execution is killed and the enforcement execution arrives as a context switch). Thus, in the schedule in the time interval $[0,D_{\mathrm{iD}}]$, there are at most $4 \cdot (\sum_{j \in \mathrm{hp(iD)} \cup \{\mathrm{iD}\}} \lceil D_{\mathrm{iD}}/T_j \rceil)$ context switches. We represent the time interval $[0,D_{\mathrm{iD}}]$ as a set of time intervals such that in each time interval, there is no context switch; we refer to each such time interval as a position. We let npos (meaning <u>n</u>umber of <u>pos</u>itions) be the sufficient number of positions needed to represent the schedule based on the above upper bound on the number of context switches; thus:

$$\mathrm{npos} = 4 \cdot (\sum_{j \in \mathrm{hp(iD)} \cup \{\mathrm{iD}\}} \lceil D_{\mathrm{iD}}/T_j \rceil) - 1$$

Recall that the number of jobs of a task τ_j with $j \in \mathrm{hp(iD)}$ is at most $\lceil D_{\mathrm{iD}}/T_j \rceil$. All of these jobs must (as stated by (0)) arrive at time 0 or later. But we permit that some of these jobs may arrive after D_{iD}; these jobs, however, are not (from (0)) allowed to perform any execution (they can be either skip jobs or they can have normal execution time being zero). We will describe the schedule with variables that are either real, integers, or Booleans. We will use a real variable to indicate time the time when an event occurs (for example a job arrival). We will use integer variables to indicate the position at which an event occurs. We will use Boolean variables to indicate whether a certain event occurs in a given position. Recall that there are five possible outcomes for a job (skip, finish in normal mode and increment counter, finish in normal mode and not increment counter, finish in enforcement mode, kill at time of absolute deadline); we will use variables to indicate whether a job has a certain outcome and this outcome was caused by an event in a certain position. We will let j be an index of a task in $\mathrm{hp(iD)} \cup \{\mathrm{iD}\}$. We will let q be an index of a job; we will let p be the index of a position.

Variables. We now state the variables, their interpretation, and their domains. The variables with the domain real numbers are the following. t^p denotes the time when position p starts. $A_{j,q}$ denotes the arrival time of $\tau_{j,q}$. $c_{j,q}$ denotes the normal execution time of $\tau_{j,q}$. $e_{j,q}$ denotes the enforcement execution time of $\tau_{j,q}$. $\mathrm{execc}^p_{j,q}$ denotes the amount of normal execution that $\tau_{j,q}$ performs in position p. $\mathrm{exece}^p_{j,q}$ denotes the amount of enforcement execution that $\tau_{j,q}$ performs in position p.

The variables with the domain integers are the following. $\mathrm{arrivespos}_{j,q}$ denote the position at which $\tau_{j,q}$ arrives. $\mathrm{finishespos}_{j,q}$ denote the position at which $\tau_{j,q}$ finishes. $\mathrm{Zexpirespos}_{j,q}$ denote the position at which $\tau_{j,q}$ Zexpires. $\mathrm{Dexpirespos}_{j,q}$ denote the position at which $\tau_{j,q}$ Dexpires. $\mathrm{counter}^p_j$ denotes the counter of τ_j in the beginning of position p. The 1^{st} position is position 1; however, we let $\mathrm{counter}^0_j$ denote the value of the counter of τ_j just before position 1 (that is, the initial value of the counter). The variables with the domain Boolean are the following. $\mathrm{respC}_{j,q}$ indicates whether $\tau_{j,q}$ is a respectC job. $\mathrm{elig}^p_{j,q}$ indicates whether $\tau_{j,q}$ is a eligible for execution (i.e., has arrived but not finished) in the beginning of position p. $x^p_{j,q}$ indicates whether $\tau_{j,q}$ executes in position p. $o_{j,q}$ is an integer (in $\{1..5\}$) stating that outcome of job $\tau_{j,q}$; we also use $op_{j,q}$ to indicate the position of the event that caused this outcome.

Constraints. We now state the constraints. Clearly, the start time of a position must be no earlier than the start time of its preceding position. Thus:

$$\forall p \in \{1..\mathrm{npos}\}. t^p \leq t^{p+1} \tag{1}$$

Also, the 1^{st} position starts at time 0. Thus:

$$t^1 = 0 \tag{2}$$

Also, the counters must initially be non-negative:

$$\forall j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}.\mathrm{counter}_j^0 \geq 0 \tag{3}$$

We now express arrival times, finishing times, Zexpiring, and Dexpiring. A job arrives in the beginning of exactly one position. We use $\mathrm{arrivespos}_{j,q}$ to indicate the position at which the job $\tau_{j,q}$ arrives. It is an integer. Clearly, it must be in the range of position indices, that is, $\{1..\mathrm{npos}+1\}$. The same applies to finishing times, Zexpiring, and Dexpiring as well. Thus:

$$\begin{aligned}&\forall\langle j,p\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}).\\ &\quad (1 \leq \mathrm{arrivespos}_{j,q}) \wedge (\mathrm{arrivespos}_{j,q} \leq \mathrm{npos}+1) \wedge (1 \leq \mathrm{Zexpirespos}_{j,q}) \wedge\\ &\quad (\mathrm{Zexpirespos}_{j,q} \leq \mathrm{npos}+1) \wedge (1 \leq \mathrm{Dexpirespos}_{j,q}) \wedge (\mathrm{Dexpirespos}_{j,q} \leq \mathrm{npos}+1) \wedge\\ &\quad (1 \leq \mathrm{op}_{j,q}) \wedge (\mathrm{op}_{j,q} \leq \mathrm{npos}+1)\end{aligned} \tag{4}$$

The time of arrival relates to the the position of arrival. Ditto for other events. Hence:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}) \wedge (p \in \{1..\mathrm{npos}+1\}).\\ &(\mathrm{arrivespos}_{j,q} = p) \Rightarrow (A_{j,q} = t^p) \wedge (\mathrm{Zexpirespos}_{j,q} = p) \Rightarrow (A_{j,q} + Z_i = t^p) \wedge\\ &\qquad (\mathrm{Dexpirespos}_{j,q} = p) \Rightarrow (A_{j,q} + D_i = t^p)\end{aligned} \tag{5}$$

There are also some bounds on execution times.

$$\begin{aligned}&\forall\langle j,q\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}).\\ &(c_{j,q} \geq 0) \wedge (\mathrm{respC}_{j,q} \Rightarrow (c_{j,q} \leq C_j)) \wedge (e_{j,q} \geq 0) \wedge (e_{j,q} \leq E_j)\end{aligned} \tag{6}$$

We also express minimum inter-arrival times as:

$$\forall\langle j,q\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil - 1\}).\ A_{j,q} + T_j \leq A_{j,q+1} \tag{7}$$

Our system model states that for RC_j consecutive jobs of τ_j, at least one is a respectC job. Thus:

$$\forall\langle j,q\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil - (\mathrm{RC}_j - 1)\}).\ \vee_{q' \in \{q..q+\mathrm{RC}_j-1\}} \mathrm{respC}_{j,q'} \tag{8}$$

We will now express constraints on the schedule based on how the scheduling is done.

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}) \wedge (p \in \{1..\mathrm{npos}\}).\\ &\qquad (\mathrm{elig}_{j,q}^p = ((\mathrm{arrivespos}_{j,q} \leq p) \wedge (p < \mathrm{op}_{j,q}))) \wedge\\ &\qquad (\mathrm{x}_{j,q}^p = (\mathrm{elig}_{j,q}^p \wedge (\wedge_{j' \in \mathrm{hp}(\mathrm{iD}} \wedge_{q' \in \{1..\lceil D_{\mathrm{iD}}/T_{j'}\rceil\}} (\neg \mathrm{elig}_{j',q'}^p))))\end{aligned} \tag{9}$$

Note that in the constraint above, we are not referring to an event but instead to the time interval of the position. Hence, the case $p = \mathrm{npos}+1$ does not need to be considered. We also express the amount of execution of a certain type (normal versus enforcement) of a job in a given position as follows:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j \in \mathrm{hp}(\mathrm{iD}) \cup \{\mathrm{iD}\}) \wedge (q \in \{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}) \wedge (p \in \{1..\mathrm{npos}\}).\\ &\qquad ((\mathrm{x}_{j,q}^p \wedge (p < \mathrm{Zexpirespos}_{j,q})) \Rightarrow (\mathrm{execc}_{j,q}^p = t^{p+1} - t^p)) \wedge\\ &\qquad (((\neg \mathrm{x}_{j,q}^p) \wedge (p < \mathrm{Zexpirespos}_{j,q})) \Rightarrow (\mathrm{execc}_{j,q}^p = 0)) \wedge\\ &\qquad ((\mathrm{x}_{j,q}^p \wedge (p \geq \mathrm{Zexpirespos}_{j,q})) \Rightarrow (\mathrm{exece}_{j,q}^p = t^{p+1} - t^p)) \wedge\\ &\qquad (((\neg \mathrm{x}_{j,q}^p) \wedge (p \geq \mathrm{Zexpirespos}_{j,q})) \Rightarrow (\mathrm{exece}_{j,q}^p = 0))\end{aligned} \tag{10}$$

Recall that $o_{j,q}$ indicates the outcome of a $\tau_{j,q}$—there are five outcomes. Also recall that $\mathrm{op}_{j,q}$ indicates position in which an event occurs for which this outcome is determined. Clearly, their ranges are as follows:

$$\begin{aligned}&\forall\langle j,q\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\}).\\ &(1\le o_{j,q})\wedge(o_{j,q}\le 5)\wedge(1\le \mathrm{op}_{j,q})\wedge(\mathrm{op}_{j,q}\le \mathrm{npos}+1)\end{aligned}\tag{11}$$

Recall that outcome 1 means that the job is skipped. Hence:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\})\wedge(p\in\{1..\mathrm{npos}\}).\\ &\qquad((o_{j,q}=1)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow\\ &\qquad(\mathrm{arrivespos}_{j,q}\le p)\wedge(p\le \mathrm{Zexpirespos}_{j,q})\wedge\\ &\qquad(\mathrm{counter}_j^{p-1}\ge 1)\wedge(\mathrm{counter}_j^{p}=\mathrm{counter}_j^{p-1}-1)\end{aligned}\tag{12}$$

We will now describe the four other possible outcomes For the case $p=1$, for $o\in\{2..5\}$ it holds that $((o_{j,q}=1)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow$ false. We will now describe the constraints for $p\ge 2$. Recall that $o=2$ represents early finishing and the counter is incremented. Thus:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\})\wedge(p\in\{1..\mathrm{npos}\}).\\ &\qquad((o_{j,q}=2)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow\\ &(\mathrm{arrivespos}_{j,q}<p)\wedge(p\le \mathrm{Zexpirespos}_{j,q})\wedge x_{j,q}^{p-1}\wedge(\sum_{p'\in\{1..p-1\}}\mathrm{execc}_{j,q}^{p'}=c_{j,q})\wedge\\ &\qquad(\mathrm{counter}_j^{p-1}<\mathrm{MAXCOUNT}_j)\wedge(\mathrm{counter}_j^{p}=\mathrm{counter}_j^{p-1}+1)\end{aligned}\tag{13}$$

Recall that $o=3$ represents early finishing and the counter is not incremented. Thus:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\})\wedge(p\in\{1..\mathrm{npos}\}).\\ &\qquad((o_{j,q}=3)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow\\ &(\mathrm{arrivespos}_{j,q}<p)\wedge(p\le \mathrm{Zexpirespos}_{j,q})\wedge x_{j,q}^{p-1}\wedge(\sum_{p'\in\{1..p-1\}}\mathrm{execc}_{j,q}^{p'}=c_{j,q})\wedge\\ &\qquad(\mathrm{counter}_j^{p-1}\ge\mathrm{MAXCOUNT}_j)\wedge(\mathrm{counter}_j^{p}=\mathrm{counter}_j^{p-1})\end{aligned}\tag{14}$$

Recall that $o=4$ represents late finishing. Thus:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\})\wedge(p\in\{1..\mathrm{npos}\}).\\ &\qquad((o_{j,q}=4)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow\\ &(\mathrm{arrivespos}_{j,q}<p)\wedge(p\le \mathrm{Zexpirespos}_{j,q})\wedge x_{j,q}^{p-1}\wedge(\sum_{p'\in\{1..p-1\}}\mathrm{execc}_{j,q}^{p'}<c_{j,q})\wedge\\ &\qquad(\sum_{p'\in\{1..p-1\}}\mathrm{exece}_{j,q}^{p'}=e_{j,q})\wedge(\mathrm{counter}_j^{p}=\mathrm{counter}_j^{p-1})\end{aligned}\tag{15}$$

Recall that $o=5$ represents finishing at deadline and the job gets killed. Thus:

$$\begin{aligned}&\forall\langle j,q,p\rangle\ (j\in \mathrm{hp}(\mathrm{iD})\cup\{\mathrm{iD}\})\wedge(q\in\{1..\lceil D_{\mathrm{iD}}/T_j\rceil\})\wedge(p\in\{1..\mathrm{npos}\}).\\ &\qquad((o_{j,q}=5)\wedge(\mathrm{op}_{j,q}=p))\Rightarrow\\ &(p=\mathrm{Dexpirespos}_{j,q})\wedge x_{j,q}^{p-1}\wedge(\sum_{p'\in\{1..p-1\}}\mathrm{execc}_{j,q}^{p'}<c_{j,q})\wedge\\ &(\sum_{p'\in\{1..p-1\}}\mathrm{exece}_{j,q}^{p'}<e_{j,q})\wedge(\mathrm{counter}_j^{p}=(\mathrm{counter}_j^{p-1}-1))\end{aligned}\tag{16}$$

If an event that creates one of these outcomes occurs, then the counter of a task should be unchanged. Thus:

$$\forall \langle j,q,p\rangle \ (j \in \text{hp}(\text{iD}) \cup \{\text{iD}\}) \wedge (q \in \{1..\lceil D_{\text{iD}}/T_j \rceil\}) \wedge (p \in \{1..\text{npos}\}). \\ (\neg(o_{j,q} = p)) \Rightarrow (\text{counter}_j^p = \text{counter}_j^{p-1}) \tag{17}$$

From (0) we obtain that there is no execution after the deadline of $\tau_{\text{iD},1}$ and no jobs arrive then. Hence, considers cannot be changed after that. Also, from (0), we obtain that before the deadline of $\tau_{\text{iD},1}$, there is no time when the counter of a task becomes negative. Hence:

$$\forall \langle j,p\rangle \ (j \in \text{hp}(\text{iD})) \wedge (p \in \{1..\text{npos}\}+1). \ \text{counter}_j^p \geq 0 \tag{18}$$

From (0) it follows that $\tau_{\text{iD},1}$ misses its deadline. Thus:

$$o_{\text{iD},1} = 5 \tag{19}$$

From (0) it follows that the initial value of the counter of τ_{iD} is zero. Thus:

$$\text{counter}_{\text{iD}}^0 = 0 \tag{20}$$

From (0) it follows that:

$$\forall p \in \{1..\text{npos}\}. \ (p \leq \text{Dexpirespos}_{j,q}) = (\vee_{j \in \text{hp}(\text{iD}) \cup \{\text{iD}\}} \vee_{q \in \{1..\lceil D_{\text{iD}}/T_j \rceil\}} x_{j,q}^p) \tag{21}$$

From (0) it follows that:

$$\forall \langle j,p\rangle \ (j \in \text{hp}(\text{iD}) \cup \{\text{iD}\}) \wedge (p \in \{1..\text{npos}\}). \ \text{counter}_j^p \leq \min(\lceil D_{\text{iD}}/T_j \rceil, \text{MAXCOUNT}_j) \tag{22}$$

We let cprsmtinstance(τ,iD) denote a function that takes a taskset τ and an integer iD as input and outputs an SMT instance; this SMT instance is constructed as given in this subsection.

4.3 Creating a schedulability condition

We will now put together the results from the two previous subsections into a schedulability condition.

▶ **Lemma 1.** *τ is not schedulable $\Rightarrow$ (0)*

Proof. Follows from the discussion in Section 4.1. ◀

▶ **Lemma 2.** *(0) $\Rightarrow$ $(\exists \text{iD} \ (\tau_{\text{iD}} \in \tau) \wedge \text{cprsmtinstance}(\tau, \text{iD}))$*

Proof. Follows from the discussion in Section 4.2. ◀

▶ **Lemma 3.** *τ is not schedulable $\Rightarrow$ $(\exists \text{iD} \ (\tau_{\text{iD}} \in \tau) \wedge \text{cprsmtinstance}(\tau, \text{iD}))$*

Proof. Follows Lemma 1 and Lemma 2. ◀

▶ **Lemma 4.** *τ is not schedulable $\Leftarrow$ $(\exists \tau_{\text{iD}} \in \tau \ \text{cprsmtinstance}(\tau, \text{iD}))$*

Proof. If the rhs is true, then $\exists \tau_{\text{iD}} \in \tau$ cprsmtinstance(τ, iD). Then we can obtain the solution and this yields an assignment R, a schedule sc, and a counter assignment ca such that R is legal, ca is legal, and sc can be generated by R and in which the 1st job of τ_{iD} misses its deadline. Hence, the lhs is true. ◀

▶ **Lemma 5.** *τ is not schedulable $\Leftrightarrow$ $(\exists \tau_{\text{iD}} \in \tau \ \text{cprsmtinstance}(\tau, \text{iD}))$*

Proof. Follows Lemma 3 and Lemma 4. ◀

▶ **Theorem 6.** *τ is schedulable $\Leftrightarrow$ $(\forall \tau_{\text{iD}} \in \tau \ \neg\text{cprsmtinstance}(\tau, \text{iD}))$*

Proof. Follows Lemma 5. ◀

4.4 Creating an algorithm for schedulability testing

Based on Theorem 1, we can now create an algorithm for schedulability testing as follows:

```
 1. flag := true
 2. for each $\tau_{\mathrm{iD}} \in \tau$ as long as flag is true do
 3.     if not sat(cprsmtinstance($\tau$, i)) then
 4.         flag := false
 5.     end if
 6. end for
 7. if flag then
 8.     stop and declare schedulable
 9. else
10.     stop and declare unschedulable
11. end if
```

5 Performance Evaluation

We have implemented a tool that performs the schedulability test presented in the previous section. Recall that this schedulability test checks satisfiability (sat) of an SMT instance. Internally, such sat solvers infer new constraints. In order to speed up the schedulability test, we will add additional (redundant) constraint that can be inferred directly. These are the following. If we consider two jobs of the same task, then we can compute a minimum inter-arrival time between them. We can also observe that in the schedule representation if a schedule exist, then we can create another representation in which there is at most one arrival, Zexpire, or Dexpire occurring. From this, we can obtain additional constraints on the integers that describe that position in which an event occurs. For example: $\mathrm{arrivespos}_{j,1} + 4 \leq \mathrm{arrivespos}_{j,2}$. With these additional redundant constraints, we implement the tool. We use the Z3 SMT solver because it is one of the most widely-used and also well-maintained SMT solvers. We will now report on the our experimental results with our tool. We begin by presenting the experimental setup (in Subsection 5.1), then present experimental results (in Subsection 5.2), and then present results from a case study (in Subsection 5.3).

5.1 Experimental setup

We implement the meta mode with our tool. We do it as follows. There are 4 experimental setup parameters (ntasks,targetutil,TMAXEXP,DTRATIO) and these are used to generate a taskset randomly as follows:

1. $|\tau|$ = ntasks
2. T_i = random(1.0, $2^{log2(TMAXEXP)}$)
3. D_i = DTRATIO $* T_i$
4. Z_i = 0.8*D_i
5. C_i = 0.9* Z_i
6. E_i = 0.1* Z_i
7. RC_i = random(1,2)
8. $\mathrm{MAXCOUNT}_i$ = 1

After all tasks have been assigned this way, we compute scalingfactor as follows scalingfactor := targetutil/$(\sum_{j=1..n} C_j/T_j)$. Then, for each task $\tau_j \in \tau$, we multipy C_j and E_j by scalingfactor.

These 4 experimental setup parameters are the following:

1. ntasks in $\{2..5\}$
2. targetutil in $\{0.2, 0.4, 0.6, 0.8.1.0, 1.2, 1.4, 1.6, 1.8, 2.0\}$
3. TMAXEXP in $\{1, 2, 4, 8, 16, 32, 64, 128\}$
4. DTRATIO in $\{0.01, 0.1, 1.0\}$

We explore all combinations of them; that is 960 combinations. For each combination, we generate one taskset and measure that time required for our tool to perform schedulability analysis. We also obtain whether the taskset is schedulable.

5.2 Experimental result

We ran the experiments on a Dell XPS x8900-8756BLK desktop computer with quad-core Intel 6700k processor and 32Gb memory running Ubuntu 17.10. We used the SMT solver Z3 (version X"'). Our tool is a C program that implement the algorithm in Section 4.4. Our C program was compiled with gcc. This program writes an SMT instance to a file and calls the SMT solver Z3. We set Z3 to verbose mode. It took 3 days to finish all the experiments

The results of the experiments are shown in Figure 3. It can be seen that (i) the time required by our new schedulability test increases grows rapidly as the number of tasks and TMAXEXP increase and (ii) at 400% utilization, there are 25% of the tasksets that are schedulable. The former indicates that there are large systems that our method is not capable of analyzing. On the other hand, the latter indicates that our method is very useful for CPS where the number of tasks is not too large.

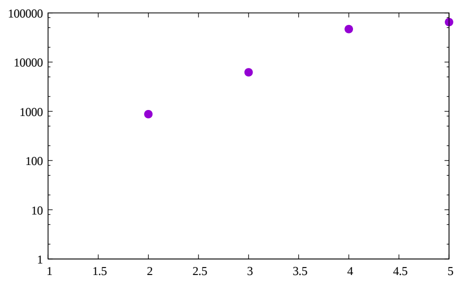
5.3 Case study

We have modeled a multi-UAV system from our previous work [1] using the task model presented in this paper. And analyzed it with the scheduling theory in this paper.

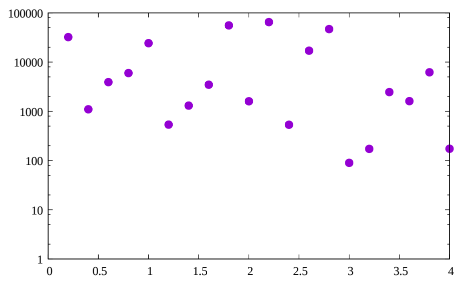
6 Related Work

We now discuss relevant previous work. These works share our goal of linking the physical world to real-time scheduling but unlike our work, they do not offer any *application-independent* way of specifying the resilience of the physical environment to deadline misses, or any schedulability analysis.

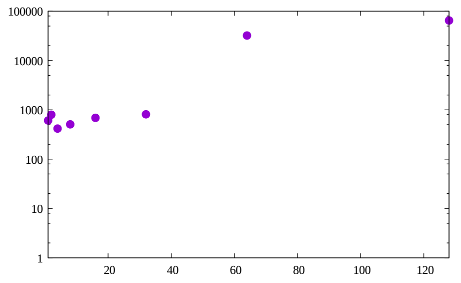
Offline selection of periods. The paper [19] considers a set of periodic feedback control tasks scheduled on a single processor. The paper mentions that the performance of a control task can be characterized using a Performance Indicator function, which is concave with respect to the controller frequency. If the Performance Indicator is small, then the performance is good. The frequency of a controller must be above a certain lower bound to prevent instability, and higher frequency leads to better performance. On the other hand, if all tasks have very high frequency, then the taskset is not schedulable. Thus, the paper formulates an optimization problem: minimize a weighted sum of the Performance Indicator function across all tasks subject to the constraint that the taskset is schedulable and that frequencies are above their minimum thresholds; here the frequencies of tasks are the decision variables. This problem of selecting sampling frequencies has also been



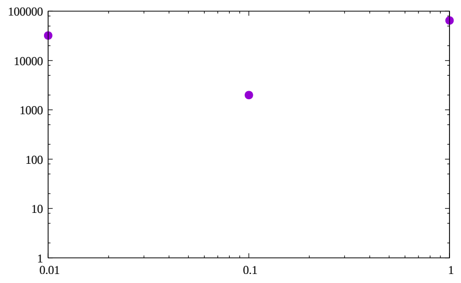
(a) Time as function of number of tasks.



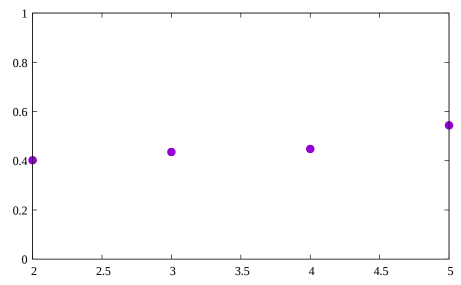
(b) Time as function of utilization.



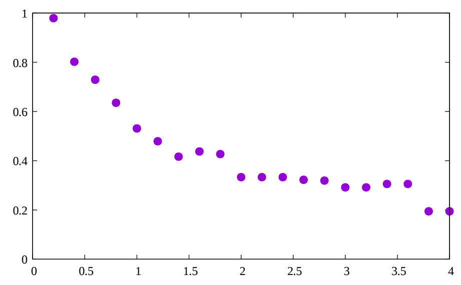
(c) Time as function of TMAXEXP.



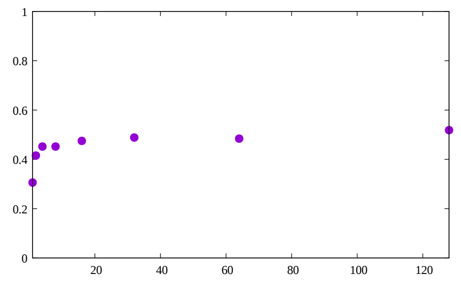
(d) Time as function of DTRATIO.



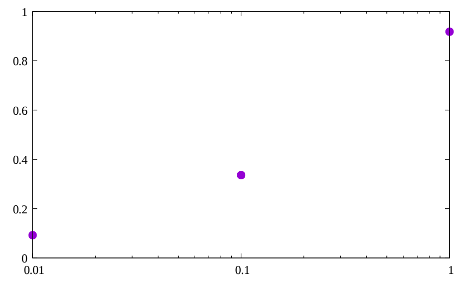
(e) Success ratio as function of number of tasks.



(f) Success ratio as function of utilization.



(g) Success ratio as function of TMAXEXP.



(h) Success ratio as function of DTRATIO.

Figure 3 Experiments on randomly generated systems. Time means time required for schedulability test to finish analysis of one system (measured in seconds).

studied in the context of surveillance radars [15]; here the problem is more complex and the paper [15] proposes to pre-compute resource allocations before run-time using templates.

Rhythmic tasks. The paper [13] considers a set of tasks scheduled with fixed-priority preemptive scheduling on a single processor. There is a physical body (a crankshaft), whose rotation is described via an angle, θ, that increases with rotation. The highest-priority task is assumed to be rhythmic, i.e., a job of this task arrives when (θ mod 2π) takes a certain value; the other tasks are periodic. The paper presents a sufficient schedulability analysis of such systems. The papers [10, 7] both extend [13]. Specifically, the paper [10] considers the rhythmic task model, a single processor, and fixed-priority scheduling. It assumes that a job arrives not only because (θ mod 2π) takes a certain value, but also that the software is adaptive in the sense that at high rotation speed (high rpm), the rhythmic task has lower execution time. The reason for lower execution time is that the task measures the rotation speed and disables certain functionalities at higher rotation speeds. The paper [7] presents a schedulability test of rhythmic tasks scheduled with EDF on a single processor.

Event-based control. The paper [3] presents an event-based PID controller that splits a control task into two pieces – one samples the physical environment and determines the need for the controller to compute a command, the other computes the command. The first piece executes periodically, and computes the absolute difference δ between the current error and the error at the last sampling. The second piece arrives if δ exceeds a certain threshold, or if a certain amount of time elapses since its last arrival. The paper shows that this event-based PID controller achieves a significant reduction of CPU consumption. One can also describe this work as periodic sampling but with state-dependent execution time.

Self-triggered control. The paper [22] presents self-triggered control. Here, a control task generates a sequence of jobs, and the arrivals are not necessarily periodic. Instead, when a job of a control task finishes, it computes the time when the next job of this task should arrive. This computation of the arrival time of the next job is based on the earliest instant in which it is necessary to perform a control computation in order to ensure stability or a control performance metric. Self-triggered control is different from the event-based control mentioned above [3] in the sense that in the event-triggered control mentioned above, tasks arrive periodically but the execution time varies (depending on whether there is a need to compute a new control action). Self-triggered control is also different from the rhythmic task model in the sense that a self-triggered control task *computes* the time of the next arrival, whereas in the rhythmical task model a task does not compute the time of its next arrival; it sleeps and wakes up based on an event whose arrival time is not known in advance. The paper [22] proposes that self-triggered control be implemented by describing the plant with a state vector (which is normal) but also adds an extra variable to the state vector and this extra variable is the currently used period of the controller. The evolution of the system is described by multiplying the current state vector with a matrix and adding a vector multiplied by a command and this yields the new state vector. Certain elements in the matrix determine how, at run-time, the next period will be computed based on the physical variables.

The paper [2] considers self-triggered control for a distributed system with computer nodes (of the type sensor, actuator, and controller) and a bus — Controller Area Networks (CAN) bus — shared between the computer nodes. The transmission times of messages between sensor-to-controller and controller-to-actuator are selected at run-time to be as late as possible without jeopardizing stability. This decision is taken based on the current state of the plants at run-time. The paper [16] presents another self-triggered controller but for a single computer node and EDF. The paper [23] extends [22] by giving explicit rules for computing the next sampling time for a controller at run-time. It also gives equations, that can be used before run-time, for computing the maximum computation demand of a controller that uses this rule. With this computed maximum computation demand, it is possible to incorporate it in standard schedulability tests for fixed-priority scheduling or EDF. The paper also shows a case study illustrating that event-triggered control can offer better control performance and less processing demand than an optimal periodic controller. Our work differs in two main aspects: (i) it preserves the decomposition of the recurrent execution and time verification using the traditional periodic task model in combination with a safety enforcer, and (ii) it allows the controller to focus on maximizing the control performance (as close as possible to C_{ϕ}^{k}) instead of staying at the limit of stability.

Mixed-criticality scheduling. Mixed-criticality scheduling [11, 4, 6] recognizes that tasks typically implement different functions; different functions may have different criticalities and different criticalities have different requirements on the confidence in its proof of correctness. Thus, it is helpful to make different assumptions about taskset parameters for different criticality levels. For example, if a task τ_i has high criticality, then when proving whether it meets deadlines, we need to have a high-degree of confidence in the parameters that are used for this proof. For example, when computing its response time, we can use worst-case execution time parameters for τ_i and for higher-priority task such that we have very high confidence that these parameters correctly provide upper bounds on execution times. On the other hand, if τ_k has lower criticality, then we do not need such a high degree of confidence. The authors in [12] present a model where jobs are allowed to miss deadlines by extending its deadline and period based on utility, which takes the form of a “variable” criticality that decreases as more CPU is given to a task (a.k.a. diminishing returns). However, this model does not take into account the execution of enforcers, or the runtime state of the physical world when missing deadlines. Mixed-criticality scheduling shares our goal of reducing pessimism, but differs by providing different taskset parameters to be used for different criticality levels whereas we describe the impact on execution of a task on the physical environment.

Skips and weakly hard real-time systems. The paper [5] presents a model called *weakly hard real-time systems*. It specifies that a task does not have to meet all its deadlines; occasional deadline misses are allowed. The task model of weakly hard real-time systems gives a precise definition of what it means that occasional deadline misses are allowed. Two examples are: (i) out of m consecutive jobs of a given task, there are at most n consecutive jobs that misses its deadline; and (ii) out of m consecutive jobs of a given task, there are at most n jobs (consecutive or not) that misses its deadline. The paper points out that there are different reasons why a deadline is missed: (i) the task is not admitted to the system so it generates no job, (ii) a job is admitted to the system but this particular job is skipped [8], (iii) a job has started to execute but it gets aborted before it finishes, and (iv) a job runs until completion but finishes after its deadline. The paper gives a schedulability test for tasks in this model scheduled with fixed-priority preemptive scheduling on a single processor. A similar model is provided by [14]. A similar problem occurs in networked control systems where some messages are lost. A scheduling approach of processors and analysis of the maximum allowed drop-out rate is provided by [18].

Run-time assurance/verification. A common technique is to monitor system states and steer a computation toward safe states. See [9] for an excellent review.

Simplex. Simplex [20] is an architecture for designing controllers. It comprises one sophisticated controller, a simple controller, a set describing the safe states, and another set which describes transitioning between controllers. The sophisticated controller is allowed to operate when the plant is the last set of states. If the plant leaves this set, then a simpler controller takes over. With this architecture, the sophisticated controller can be optimized for performance and does not need to be verified; the simple controller, however, is verified to make sure that the plant is always in a safe state. One can think of the simple controller in Simplex as somewhat analogous to our enforcer execution (E_i of task τ_i). Simplex, however, does not consider real-time scheduling.

Common to the above-listed previous work is that none of them offers a model with associated schedulability analysis of real-time tasks where the physical environment imposes a requirement on the interaction with the software and describe this in an application-independent way. In this paper, we have presented such a model and an exact schedulability test for it.

7 Conclusions

We have presented a new task model and test to verify the schedulability of the resilience of a cyber-physical system. The schedulability test is exact and it is based on solving a sequence of SMT instances. A key benefit of this scheduling theory is that it allows tasksets with 400% to be guaranteed.

Acknowledgement. Copyright 2018 Carnegie Mellon University and Mentor Graphics. All Rights Reserved. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT. [DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM18-0092.

References

1 B. Andersson, S. Chaki, and D. de Niz. Combining symbolic runtime enforcers for cyber-physical systems. In *RV*, 2017.

2 A. Anta and P. Tabuada. On the benefits of relaxing the periodicity assumption for networked control systems over CAN. In *RTSS*, 2009.

3 K.E. Årzén. A simple event-based PID controller. In *IFAC World Congress*, 1999.

4 S. Baruah. Schedulability analysis for a general model of mixed-criticality recurrent real-time tasks. In *RTSS*, 2016.

5 G. Bernat, A. Burns, and A. Llamosí. Weakly hard real-time systems. In *IEEE Transactions on Computers*, 2001.

6 A. Burns and R. Davis. Mixed criticality systems - a review. http://www-users.cs.york.ac.uk/burns/review.pdf.

7 G. C. Buttazzo, E. Bini, and D. Buttle. Rate-adaptive tasks: Model, analysis, and design issues. In *DATE*, 2014.

8 M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *RTSS*, 1997.

9 M. Clark, X. Koutsoukos, R. Kumar, I. Lee, G. Pappas, L. Pike, J. Porter, and O. Sokolsky. A study on run time assurance for complex cyber physical systems. In *Technical Report*, 2013.

10 R. I. Davis, T. Feld, V. Pollex, and F. Slomka. Schedulability tests for tasks with variable rate-dependent behaviour under fixed priority scheduling. In *RTAS*, 2014.

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

12 D. de Niz, L. Wrage, A. Rowe, and R. Rajkumar. Utility-based resource overbooking for cyber-physical systems. *ACM Trans. Embed. Comput. Syst.*, 2014.

13 J. Kim, K. Lakshmanan, and R. Rajkumar. Rhythmic tasks: A new task model with continually varying periods for cyber-physical systems. In *ICCPS*, 2012.

14 P. Kumar and L. Thiele. Quantifying the effect of rare timing events with settling-time and overshoot. In *RTSS*, 2012.

15 C. Lee, C.-S. Shieh, and L. Sha. Online QoS optimization using service classes in surveillance radar systems. In *JRTS*, 2004.

16 M. Lemmon, T. Chantem, X. Hu, and M. Zyskowski. On self-triggered full information H-infinity controllers. In *Hybrid Systems: Computation and Control*, 2007.

17 W. Lucia, B. Sinopoli, and G. Franzè. A set-theoretic approach for secure and resilient control of cyber-physical systems subject to false data injection attacks. In *SOSCYPS*, 2016.

18 I. Saha, S. Baruah, and R. Majumdar. Dynamic scheduling for networked control systems. In *HSCC*, 2015.

19 D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control systems. In *RTSS*, 1996.

20 L. Sha. Using simplicity to control complexity. *IEEE Software*, 2001.

21 L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. In *proceedings of the IEEE*, 1994.

22 M. Velasco, J. Fuertes, and P. Martí. The self triggered task model for real-time control systems. In *RTSS-WIP*, 2003.

23 M. Velasco, P. Martí, and E. Bini. Control-driven tasks: Modeling and analysis. In *RTSS*, 2009.