Schedulability Analysis and Software Synthesis for Graph-Based Task Models with Resource Sharing

Jakaria Abdullah*, Morteza Mohaqeqi*, Gaoyang Dai* and Wang Yi*
*Department of Information Technology
Uppsala University, Uppsala, Sweden

***Abstract*—Currently the main approaches to model-based design of embedded software rely on the synchronous paradigm where the executions of software components are either statically ordered or enforced using predefined orderings e.g. Simulink diagrams [1]. However, these approaches may result in resource over provisioning and inflexibility e.g. adding a new function block may require re-designing the whole system.**

To overcome these drawbacks, we use a dynamic approach allowing multi-tasking implementation of software components using real-time tasks. The challenge is run-time scheduling and schedulability analysis of real-time tasks with inter-task communication (i.e. resource sharing). In this paper, we use a graph-based task model (DRT [2] developed in previous work) to describe software components as a system of real-time tasks sharing not only a uniprocessor but also non-preemptive resources e.g. accesses to shared data. However, timing analysis for such general task model with mixed execution of preemptive and non-preemptive jobs is yet to be developed. As the main technical contribution, we present an exact schedulability test for task systems containing both preemptive and non-preemptive computation jobs with experimental evaluations showing the efficiency of our approach for realistic workload such as the engine control applications. We also present an approach to generate event-triggered Ada programs from analyzed design models.

I. Introduction

Model-based design (MBD) has become an established approach for developing embedded software. The basic idea of this approach is to use formal models throughout the development cycle, from design, to analysis, to implementation. Existing MBD solutions, such as Simulink[1] use Synchronous Block Diagrams (SBDs) [33] as the model of computation. The basic component in an SBD is a block, which can be a state machine (known as Stateflow [25] in Simulink) with inputs and outputs to be used for designing controller software. The external events that can work as inputs are assumed to be generated by periodic signals and outputs of blocks can be connected to inputs of other blocks to form a diagram. The execution of a block corresponds to a local reaction step of the corresponding state machine: the machine reads its local inputs, computes its local outputs and updates its local state. The topological dependency among the blocks defines a partial order relation which is maintained during execution within a synchronous step.

To generate executable code from the model, Simulink uses a step interval with a fixed (positive) size, which is obtained based on the user-specified "base rate". Every block has a sample time or period, which should be an integer multiple of the "base rate". A *single task* implementation of the model executes all the blocks maintaining their partial order relations to avoid data inconsistency. If the model is multi-rate, where different parts are executed at different periods, the *single task* implementation should run at the fastest possible period. As a result, the execution of the longest chain of blocks should fit within one period of the base rate. This requirement can significantly reduce schedulable utilization of the system and also reduces flexibility of the implementation [34].

Multi-task implementation can provide improvement in utilization by mapping blocks to different tasks and use preemptive scheduling. However, tasks now need to communicate with each other if their mapped blocks exchange data [34]. This is equivalent to solving resource sharing problem of multi-tasking as the tasks can communicate using shared data. Existing *multi-task* implementations such as [35], [36] focus on using periodic tasks and lock-free inter-task communication. However, a periodic task is not well-suited to model state-dependent behavior as execution behavior of all states should be captured by single job type, which may reduce schedulability.

Modeling state-dependent task execution behavior is the key motivation behind graph-based real-time task models [3]. The Digraph Real-Time (DRT) task model [2] is an expressive graph-based model with efficient timing analysis for independent tasks [21]. In this paper, we choose DRT as a representative of graph-based models for *multi-task* implementation. So to handle inter-task communication, DRT tasks need to be extended with resource sharing.

To solve resource sharing problem of DRT tasks, we propose non-preemptive execution of a job accessing shared data. Non-preemptive execution not only simplifies mutual exclusion in shared data access but also reduces preemption related overheads and minimizes delay in control applications [6]. However, all the jobs of a task may not use shared data and it is reasonable to use non-preemptive execution only during shared data access. As a result, DRT tasks will have both preemptive and non-preemptive jobs. Existing timing analysis methods for DRT tasks can handle either fully preemptive [21] or fully non-preemptive jobs [22]. To allow mixed execution of preemptive and non-preemptive jobs in DRT, we need to develop a new timing analysis method.

Timing analysis for non-preemptive execution is known to

[1]Simulink is a Registered Trademark of The MathWorks, Inc. [1].

be challenging even for periodic tasks [17] due to a scheduling anomaly, known as *self-pushing*. *Self-pushing* occurs when high priority jobs released during non-preemptive execution of a job are pushed ahead to interfere successive jobs. As a result, it is difficult to find the worst-case interference suffered by a job. *Self-pushing* also effects timing analysis of graph-based tasks such as DRT, as shown in the case of fully non-preemptive jobs [22]. In the setting of this paper, such analysis has additional difficulty because tasks can now release both preemptive and non-preemptive jobs by following different paths of its graph.

Our main technical contribution in this paper is an exact fixed priority uniprocessor schedulability test for DRT tasks containing both preemptive and non-preemptive jobs. To achieve this, we first define the concept of time interval during which the processor is continuously busy. Such intervals (also known as busy window) are utilized in constructing worst-case execution scenarios for DRT jobs. The main challenge here is to predict the exact size of the busy window leading to the maximal interference for a job. We overcome this problem by (a) systematically extending the scheduling window to construct a busy window that maximizes interference and (b) using workload abstractions in computing interferences. Interestingly, we need different schedulability tests for preemptive and non-preemptive jobs as these require different workload abstractions and busy window constructions. As a result, our test is a generalization of the previous schedulability tests for fully preemptive [21] and fully non-preemptive DRT tasks [22].

An experimental evaluation of our method shows that the average analysis run-time is below 10 seconds for task sets with maximum 25 tasks and 55% utilization. As potential application, we show a fully non-preemptive engine controller task of 23% utilization is scehdulable even with additional resource sharing DRT task set of more than 10% utilization. This is useful considering the fact that usually engine controller task run alone in a processor due to their complex release behavior. As a final contribution, we provide a code generation method in Ada to implement event-triggered job release and non-preemptive resource sharing with DRT model.

The rest of the paper is organized as follows. First we review previous related work on timing analysis of non-preemptive scheduling, resource sharing and code generation in Section II. Section III introduces the task model considered in this work. Our proposed exact schedulability analysis method is described in Section IV. The proposed method is evaluated in Section V. In Section VI, we illustrate a way to implement non-preemptive resource sharing for DRT in Ada using protected object. Finally, we conclude the paper with a summary together with future works in Section VII.

II. Related Work

Semantics-preserving implementation from SBD models is possible using a simple read-compute-write loop for single-rate system models. Commercial code generators like Simulink embedded coder [35] and SCADE KCG [36] provide periodic and sporadic task synthesis from both normal and Synchronous Finite State Machine (FSM) blocks which is highly inefficient in terms of resource utilization [32]. Periodic multi-task implementation is optimized in [43] by partitioning the FSMs based on the periods of their trigger events. Preservation of synchronous semantics is difficult [32] in the case of multi-rate models where multi-task implementation is more eligible. Several works (e.g., [34], [41]) propose lock-free sample-and-hold buffering to ensure correct semantics at the price of memory overhead and delays.

The logical execution time (LET) [28] concept assumes I/O as zero execution time activity which is performed at the start and the end of the task execution. Such task-level timed I/O semantics is also used in the Timed Multitasking Ptolemy environment [31] and in TrueTime toolbox [30]. However, this semantics increases delay in data reading as a task can finish computation of data long before the deadline of the current job. An emerging model based design approach is to separate development environment for the plant and the controller and use co-simulation, such as [29]. However, even in this approach controller tasks have to share resource for data transfer and synchronization.

Schedulability analysis techniques for periodic tasks using different resource sharing protocols in uniprocessor are presented in [4]. In case of graph-based task models, Jeffay et al. [10] proposed a resource sharing protocol called the Deadline Dynamic Modification (DDM) for sporadic task with multiple sequential execution phases. Ekberg et. al [11] provided the feasibility analysis of an optimal resource sharing protocol for the Generalized Multiframe (GMF) [5] task model. Both of these protocols [10], [11] are for dynamic priority based scheduling.

An alternative solution to the problem of using complex buffer management or resource sharing protocols to maintain data consistency is to use non-preemptive execution while accessing the data. This eliminates possibilities of data corruption due to the preemption from high priority tasks while a low priority task is accessing the data. Non-preemptive resource access also simplifies the situation of nested access to shared resources as a job can not be preempted while accessing the shared data.

Non-preemptive scheduling of periodic tasks has been analyzed for CAN bus [17]. George et. al [18] provided a general analysis of fully non-preemptive scheduling for periodic tasks. In case of task models that are not periodic, DRT has a feasibility analysis for fully non-preemptive scheduling [22].

Mixed execution of preemptive and non-preemptive jobs is known as limited preemptive scheduling in real-time system literature. Buttazzo et al. in [6] summarized existing limited preemptive schedulability analysis methods for periodic and sporadic tasks. Baldovin et al. [7] explored different implementation strategies to execute a critical section using fixed priority limited preemptive scheduling of periodic tasks. To the best of our knowledge, limited preemptive scheduling of graph-based task models is only considered for the Sporadic DAG task model [19] in multiprocessor settings [8]. Intra-job

parallelism inherent in DAG model is of limited application in uniprocessor while the task model considered by us can capture complex job-level release patterns even in uniprocessor. In future, we will consider an extension of DRT with Fork-Join feature [20] to apply our proposed methods in multiprocessor.

Busy window based timing analysis is originally proposed by Lehoczky [12] and extended by Tindell [9] for periodic tasks executing in uniprocessor. In [14], authors applied busy window technique for sequential task graphs executing preemptively and communicating non-preemptively using CAN bus messages. MAST [37] tool provides timing analysis of object-oriented real-time system designed by MARTE UML [38] implementing offset based response time analysis [15]. However, the analysis in both [37] and [14] apply only for acyclic or tree-shaped task graphs. Kurtin et. al [16] proposed a timing analysis for cyclic task graphs combining precedence constraints and offsets. Existing timing analysis techniques on task graphs have three major differences with our approach. Firstly, existing work focuses on meeting end-to-end deadlines where our method tries to meet individual deadlines of jobs in a task graph. Secondly, majority of the existing work focuses on multiprocessor platforms while our analysis is for uniprocessor. Finally, our method is valid for arbitrary task graphs while existing techniques mostly focus on acyclic task chains.

III. Task Model

In this section, we review the syntax and semantics of the DRT task model. In addition, we introduce non-preemptive jobs in the context of DRT model that will be used in the analysis presented in subsequent sections.

A. Syntax

A DRT task T is described by a directed graph $G(T) = (V, E)$, where V and E denote the set of vertices and edges, respectively. Each vertex $v \in V$ represents a *job type*, which is characterized by a worst-case execution time (WCET), denoted by $e(v)$, and a relative deadline, denoted by $d(v)$. It is assumed that $e(v)$ and $d(v)$ are positive integers. Each instance of a job type is called a *job*. At runtime, a task releases a sequence of jobs. Each job needs to finish its execution not later than its deadline. In this case, the job is said to be schedulable.

Each edge $(v, u) \in E$ is labeled by a positive integer, denoted by $p(v, u)$, which specifies the minimum inter-release separation time between the respective jobs. The absence of an edge between v and u means that a job of type u cannot be released directly after a job of type v. Multiple edges outgoing from a vertex show a non-deterministic choice for the type of the next job. A job type $v \in V$ has a *constrained* deadline if for all $(v, u) \in E$ it holds that $d(v) \leq p(v, u)$. Throughout this work, we assume that all deadlines are constrained.

In addition to the original syntax [2] of DRT task, we annotate a job type $v \in V$ to be either preemptive or non-preemptive. Assuming τ as a set of DRT tasks, we denote the set of all non-preemptive job types of τ by $\mathsf{NPR}(\tau)$. This

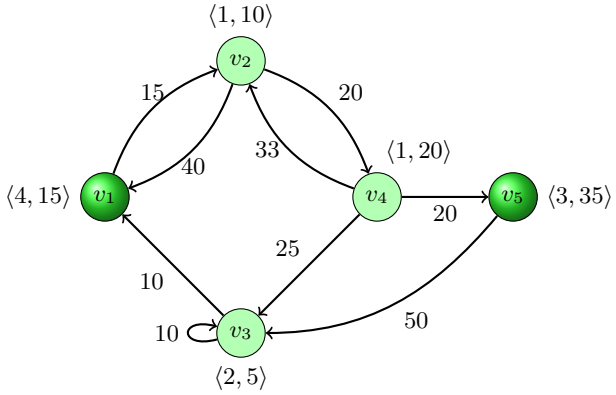


Fig. 1: A DRT task with five vertices. The shaded job types v_1 and v_5 are non-preemptive.

notation generalizes fully preemptive and fully non-preemptive DRT task sets. A fully preemptive task set has $\mathsf{NPR}(\tau) = \emptyset$, while the latter one is achieved by setting $v \in \mathsf{NPR}(\tau)$ for all $v \in V(T)$ for any $T \in \tau$.

We assume that each task T is assigned a fixed priority which is specified by a unique number, denoted by π_T. A lower value of π_T shows a higher priority. During the execution of a preemptive job, if a higher priority job is released, the running job is suspended, and the newly arrived one will get the processor. In contrast, once a non-preemptive job starts its execution, it continues until it is finished.

Figure 1 illustrates a sample DRT task with five vertices and nine edges. The shaded jobs v_1 and v_5 are non-preemptive, while the rest of the jobs are preemptive.

B. Semantics

The semantics of a DRT task is captured by the job sequences it can generate. A job is specified by a triple (R, C, v), where R, C, and v, respectively, denote the corresponding release time, execution time, and job type. A job sequence is defined as a (possibly infinite) sequence of jobs. A DRT task T described by $G(T) = (V, E)$ can generate the job sequence $[(R_1, C_1, v_1), (R_2, C_2, v_2), \ldots]$ if $v_1 \to v_2 \to \ldots$ is a path in $G(T)$, and

- $R_{i+1} - R_i \geq p(v_i, v_{i+1})$, for all $i \geq 1$; and
- $C_i \leq e(v_i)$, for all $i \geq 1$.

Note that the release times of two consecutive jobs in a job sequence are only separated by a lower bound, thus subassuming all types of *sporadic* job release behavior. On the other hand, the execution time is upper bounded by the WCET of the corresponding job. For non-preemptive job execution, this is similar to the *floating model* of *Deferred preemption scheduling* [6] where the maximum length of the non-preemptive region is specified with the task model.

IV. Schedulability Analysis

In this section, we present our exact schedulability test for DRT tasks with mixed preemptive and non-preemptive job execution. First we define the notion of busy window and the

functions that abstract DRT workload in any interval. Then we construct necessary and sufficient condition for a DRT job to be schedulable in a given release scenario. In the next step, we extend the schedulability condition to be usable in the worst-case release scenario and provide relevant proofs. Finally, we present a pseudocode of our schedulability test.

Assume a task set τ with a priority assignment Q. We want to test whether jobs of job type $v \in G(T)$ of a task $T \in \tau$ will always be able meet their deadlines. Let $\tau_{<T}$ and $\tau_{>T}$ denote the set of tasks with higher and lower priority than T respectively. Now we review some observations regarding non-preemptive execution which may effect schedulability analysis:

A. Blocking and Busy Window

Unlike fully preemptive scheduling, a higher priority job may be delayed by non-preemptive execution of a lower priority job that starts before it. To incorporate this blocking effect into schedulability analysis, we define the maximum blocking time from a lower priority task experienced by a job of job type $v \in G(T)$ as

$$B(\tau_{>T}) \equiv \max \{e(v') \mid v' \in \mathsf{NPR}(\tau_{>T})\}. \tag{1}$$

With non-preemptive execution, the interference that a job experiences also depends on the jobs that execute in the continuously busy interval before the concerned job's release [13],[22]. This concept of continuously busy interval, also known as *busy window* is inspired by Lehoczky's notion of busy period [12]. We formally define a busy window in the context of mixed preemptive non-preemptive job execution as following:

Definition 1 (Busy Window): Consider a job J, released at time t_r, with a given release scenario for other jobs (released by the J's task or by other tasks). We define t_1 and t_l as:

$$t_1 \equiv \max \{t \leq t_r \mid \text{all the jobs with a priority higher than or equal to } J \text{ released before } t \text{ have finished their execution by (i.e., before or exactly at) t}\}, \tag{2}$$

$$t_l \equiv \max \{t < t_r \mid t \text{ is the start time of a non-preemptive lower priority job (than } J)\}. \tag{3}$$

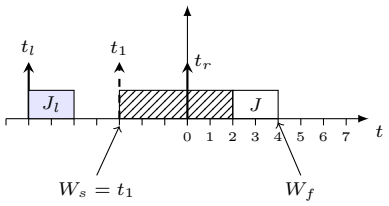
We denote the job that starts at t_l by J_l, and its WCET by e_l. In particular, if J_l does not exist, $t_l = -\infty$, $e_l = 0$. The busy window of J is defined as the interval $[W_s, W_f]$, where

$$W_s = \begin{cases} t_1, & \text{if } t_1 \geq t_l + e_l, \\ t_l, & \text{otherwise,} \end{cases} \tag{4}$$

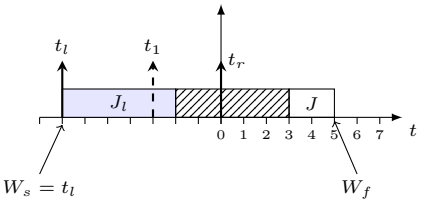
and W_f is the finish time of J.

Figure 2 demonstrates the busy window of a job J in two possible cases. In the scenario depicted in Fig. 2a, the busy window does not include any lower priority job. In contrast, Fig. 2b illustrates a situation where the busy window begins with the execution of a non-preemptive lower priority job.

Remark 1: According to the definition of t_1 in (2), the processor is continuously busy during $[t_1, t_r]$.



(a) Illustration of the first case in (4), i.e., where $t_1 \geq t_l + e_l$.



(b) Illustration of the second case in (4), i.e., where $t_1 < t_l + e_l$.

Fig. 2: A schematic view of busy window. (Hatched area depicts the execution of higher priority workload than J.)

The definition of busy window in (1) entails continually activeness of the processor during $[W_s, W_f]$, as shown by the following lemma.

Lemma 1: During the interval $[W_s, W_f]$, the processor is continually busy with job execution.

Proof 1: We first show that the processor cannot be idle during $[W_s, t_r]$. For this, we inspect two possible cases of W_s in (4). If the first condition in (4) holds, which implies $W_s = t_1$, then according to Remark 1, the processor is busy in $[W_s, t_r]$. On the other side, if $t_1 < t_l + e_l$, which means $W_s = t_l$, then the processor must be busy with executing J_l during $[W_s, W_s + e_l]$. As it is also busy during $[t_1, t_r]$ (based on Remark 1), and since $t_1 < W_s + e_l$, the processor is continuously busy during $[W_s, t_r]$. Next, we observe that the processor must be busy during $[t_r, W_f]$. Otherwise, J would be completed at some instant before W_f, which contradicts the definition of W_f. Putting these two results together, the processor is always busy in $[W_s, W_f]$.

According to the definition of busy window, all higher and equal priority jobs than J that are released before W_s must have finished their execution by W_s. In addition, no lower priority job (than J) is executed after J_l starts executing during the busy window (due to the definition of J_l). Based on these observations, the following proposition and lemma follow.

Proposition 1 (Higher Priority Interference): Let $[W_s, W_f]$ denote a busy window of a job J. Then, among the jobs with an equal or higher priority than J, only those that are released in $[W_s, W_f]$ are executed in $[W_s, W_f]$.

Lemma 2: In a busy window of a job J as defined in Definition 1, from the lower priority tasks, either no job is executed, or one non-preemptive job is executed.

Proof 2: We first notice that no preemptive lower priority job can be executed during the busy window. To observe this, based on (4) we consider two possible cases with respect to

(the beginning of) the busy window. In the first case, i.e., when $t_1 \geq t_l + e_l$, there is always some workload in the system during the busy window with the same or higher priority of J; hence no preemptive lower priority job can find a chance of execution. In the second case, i.e., when $W_s = t_l$, the same situation holds for the time interval $[t_1, W_f]$. Moreover, in $[W_s, t_1]$, the non-preemptive job J_l is executing (according to the definition of t_l and since $W_s = t_l$). As a result, again there is no chance for a preemptive lower priority job to be executed during $[W_s, W_f]$.

Next, we show that at most one lower priority job may be executed in a busy window. According to the definition of J_l, we only need to show that no lower priority job can start before W_f. As t_r is the release time of J, after t_r (i.e., during the interval $[t_r, W_f]$), J is always privileged for execution over any (not started) lower priority job, which completes the proof.

B. Workload Abstraction

We use the notion of *request function* to capture the workload generated by DRT task in an interval, as defined below.

Definition 2 (Request Function, [22]): Consider a DRT task T described by a graph $G(T)$. Let $\pi = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_l$ denote a path in $G(T)$. The request function associated to π, defined over the domain of non-negative numbers, is described as

$$rf_\pi(t) \equiv \max_{1 \leq j \leq l} \left\{ \sum_{1 \leq k \leq j} e(v_k) \mid \sum_{1 \leq k < j} p(v_k, v_{k+1}) < t \right\} \quad (5)$$

As a special case, we have $rf_\pi(0) = 0$. Intuitively, $rf_\pi(t)$ denotes the maximum workload that T can release within any time interval of length t through π or any prefix of π. We also define the notion of *inclusive* request function as follows.

Definition 3 (Inclusive Request Function, [22]): Assume that $\pi = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_l$ denotes a path in $G(T)$ for a DRT task T. The inclusive request function associated to π is then defined as

$$rf^\bullet_\pi(t) \equiv \max_{1 \leq j \leq l} \left\{ \sum_{1 \leq k \leq j} e(v_k) \mid \sum_{1 \leq k < j} p(v_k, v_{k+1}) \leq t \right\} \quad (6)$$

Based on this definition, $rf^\bullet_\pi(t)$ captures the maximum workload that can be released in any interval of length t, *including* the execution time of the job released exactly at the ending point of the interval. In particular, this implies $rf^\bullet_\pi(0) = e(v_1)$.

For a task T, we denote the set of request functions associated with the paths in $G(T)$ by $RF(T)$. Further, for a task set $\tau = \{T_1, \ldots, T_N\}$, we denote all combinations of the tasks' request functions by $RF(\tau)$; that is, $RF(\tau) \equiv RF(T_1) \times \ldots \times RF(T_N)$. We also denote elements of $RF(\tau)$ by $\bar{rf} = (rf^{(T_1)}, \ldots, rf^{(T_N)})$ where $rf^{(T)}$ denotes a request function of task T. Analogously, we use $RF^\bullet(T)$ and $RF^\bullet(\tau)$ to denote those of *inclusive* request functions.

Additionally, we require a notion for capturing the maximum workload that a task can release through any *suffix* of a path. For this, we define *suffix* request function as follows.

Definition 4 (Suffix Request Function, [22]): Let $\pi = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_l$ denote a path in the graph $G(T)$ of a DRT task T. The suffix request function associated to π is defined as

$$rf^{sfx}_\pi(t) \equiv \max_{1 \leq j \leq l} \left\{ \sum_{j \leq k \leq l} e(v_k) \mid \sum_{j < k \leq l} p(v_{k-1}, v_k) \leq t \right\} \quad (7)$$

We also denote the set of suffix request functions of all paths leading to a certain vertex v by $RF^{sfx}(v)$.

C. Analysis for a Concrete Case

In this section, we specify a necessary and sufficient condition for the job J to meet its deadline in the assumed release scenario (or equivalently, in the respective busy window). For presentation purposes, we define $x \equiv t_r - W_s$, referred to as the *busy window extension* [22].

We assume that T denotes the task releasing J, and v is the job type of J. Let $\tau_{<T}$ denote the set of tasks with a priority higher than T. Also, let $wl^{\bullet(T')}(x+t)$ denote the total workload, i.e., total execution time of the jobs, released by a task T' in the interval $[W_s, W_s + x + t]$, for $t \geq 0$. Further, let $wl^{(T')}(x+t)$ denote the workload specified by $wl^{\bullet(T')}(x+t)$ subtracted by the execution time of the job released at W_s+x+t, if any. We call $wl^{\bullet(T')}(x+t)$ the inclusive workload, and $wl^{(T')}(x+t)$ the exclusive workload. Then, the overall inclusive workload released by higher priority tasks than T in the interval $[W_s, W_s + x + t]$, for any $t \geq 0$, can be written as $wl^{\bullet hp}(x+t) = \sum_{T' \in \tau_{<T}} wl^{\bullet(T')}(x+t)$. Similarly, the total exclusive workload is computable by $wl^{hp}(x+t) = \sum_{T' \in \tau_{<T}} wl^{(T')}(x+t)$. Based on this, the total inclusive workload which may be executed during $[W_s, W_s + x + t]$, excluding J's execution time, is obtained by

$$wl^\bullet(t) = \begin{cases} wl^{sfx}(x) - e(v) + wl^{\bullet hp}(x+t), & \text{if } t_1 \geq t_l + e_l, \\ wl^{sfx}(x) - e(v) + wl^{\bullet hp}(x+t) + e_l, & \text{otherwise,} \end{cases} \quad (8)$$

where $wl^{sfx}(x)$ denotes the workload released by T during $[W_s, R]$ (including that of J). Also, the exclusive workload is computed by

$$wl(t) = \begin{cases} wl^{sfx}(x) - e(v) + wl^{hp}(x+t), & \text{if } t_1 \geq t_l + e_l, \\ wl^{sfx}(x) - e(v) + wl^{hp}(x+t) + e_l, & \text{otherwise.} \end{cases} \quad (9)$$

Based on these definitions, we provide a schedulability test for the job J. For this, we note that J is either preemptive or non-preemptive. If J is non-preemptive, then, it is schedulable if, and only if, there exists a non-negative value $t \leq d(v) - e(v)$ such that

$$wl^\bullet(t) \leq x + t. \quad (10)$$

The intuition behind the test is that, if Condition (10) holds for some $t \geq 0$, then, according to Lemma 1, the processor must have executed all the workload with a possible interference with J up to time t; thus, J starts its execution in some point in $[0, t]$. Since $t \leq d(v) - e(v)$, and also as J is non-preemptive,

J will be completed by its deadline. Besides, if no such t is found, then, during $[W_s, d(v) - e(v)]$ there is always some workload more eligible than J to execute, leading J to miss its deadline.

Besides, if the job J is preemptive, then it is schedulable if, and only if, there exists a positive $t \leq d(v)$ such that

$$wl(t) + e(v) \leq x + t. \tag{11}$$

The test examines the existence of an instant t such that the workload interfering with J released before t, as well as the workload of J, has been completely executed.

D. Generalization

The above analysis is specific to an assumed release scenario. In the next step, we explore that what the *worst-case* release scenario, or the so-called *critical instant*, is, when assuming a fixed busy window extension x.

Our analysis for the general case is based on the following observations with respect to the busy window of a job J.

- For any busy window $[W_s, W_f]$ that does not include a lower priority job J_l, i.e., whenever $t_1 \geq t_l + e_l$, there exists a busy window w.r.t another scenario that includes the execution of a lower priority job and leads to a *worse* situation for job J, if a lower priority job J_l exists, i.e., $B(\tau_{>T}) > 0$. Such a scenario can be constructed by letting the lower priority job start execution right before W_s, and keeping the arrival pattern of the higher priority jobs during $[W_s, W_f]$ intact. From (8) and (9), it is seen that, in this latter scenario, the interfering workload, i.e., $wl(t)$ and $wl^{\bullet}(t)$, which is written in the LHS of (10) and (11), in the worst case, is increased by the WCET of the lower priority job, for any $t \geq 0$. This reveals a more stringent condition for schedulability of J.
- In a busy window $[W_s, W_f]$ that starts with the execution of a lower priority job, the interfering workload of any higher priority task T' is maximized when T' releases a job right after W_s. Accordingly, in the worst case, the maximum workload of such a task up to any time $t \geq 0$ is captured by $rf^{(T')}(x + t)$, where $rf^{(T')}(.)$ denotes the respective inclusive or exclusive request function of T'.

Putting it together, we provide a schedulability test for a job J under all scenarios that reveal a busy window extension of size x.

Lemma 3 (Sufficient Schedulability Test for a Fixed Busy Window Extension): A job of type v released by a DRT task T is schedulable in any release scenario which causes a busy window extension of size x if

$$\forall \bar{rf} \in RF^{\bullet}(\tau_{<T}), rf^{sfx} \in RF^{sfx}(v) : \exists t \leq d(v) - e(v) : \\ B(\tau_{>T}) + rf^{sfx}(x) - e(v) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x + t) \leq x + t, \tag{12}$$

when $v \in \mathsf{NPR}(\tau)$, and

$$\forall \bar{rf} \in RF(\tau_{<T}), rf^{sfx} \in RF^{sfx}(v) : \exists t \leq d(v) : \\ B(\tau_{>T}) + rf^{sfx}(x) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x + t) \leq x + t, \tag{13}$$

otherwise.

Proof 3: We first show that when $v \in \text{NPR}(\tau)$, then if there exists a t satisfying Condition (12), the job is schedulable. According to Proposition 1 and Lemma 2, during time interval $[W_s, W_s + x + t]$, the processor is busy no more than $b = B(\tau_{>T}) + rf^{sfx}(x) - e(v) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x + t)$ time units. According to Lemma 1, the processor is busy during $[W_s, t_r]$, which means that it executes exactly x time units of the workload in this interval. Thus, the remaining workload is bounded by $b - x$, which, when (12) holds, satisfies $b - x \leq t$. This means that the workload to be executed/scheduled by the processor in $[t_r, t_r + t]$ is not more than t time units. Therefore, the processor must become idle in some point in $[t_r, t_r + t]$, where it will start to execute the job v. Consequently, the start time of v is no later than $t_r + t$. Since $t \leq d(v) - e(v)$, the job v will finish its execution by its deadline in this situation.

When $v \notin \mathsf{NPR}(\tau)$, the workload released by higher priority tasks *after* the start of J can also delay its finish time. If Condition 13 holds, we are sure that there exists a time instant no later than the deadline of J at which all interfering workload, as well as the J's workload are computed. This means that J meets its deadline.

Based on Lemma 3, if the schedulability condition holds for all possible values of x, then the job turns out to be schedulable. We notice that x is upper-bounded by a value L, where L is the smallest t satisfying $\sum_{T \in \tau} mrf^{(T)}(t) \leq t$ [22], in which

$$mrf^{(T)}(t) \equiv \max \{rf(t) \mid rf(.) \in RF(T)\} \tag{14}$$

Using this, we present a sufficient test for schedulability of a job.

Lemma 4 (Sufficient Schedulability Condition): Consider a job J of type v that is released by task T. If $v \in \text{NPR}(\tau)$, then J is schedulable if

$$\forall x \leq L, \bar{rf} \in RF^{\bullet}(\tau_{<T}), rf^{sfx} \in RF^{sfx}(v) : \exists t \leq d(v) - e(v) : \\ B(\tau_{>T}) + rf^{sfx}(x) - e(v) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x + t) \leq x + t. \tag{15}$$

In addition, if $v \notin \text{NPR}(\tau)$, then J is schedulable if

$$\forall x \leq L, \bar{rf} \in RF(\tau_{<T}), rf^{sfx} \in RF^{sfx}(v) : \exists t \leq d(v) : \\ B(\tau_{>T}) + rf^{sfx}(x) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x + t) \leq x + t. \tag{16}$$

Now, we argue that the provided test serves as a *necessary* condition for schedulability of a job J, too. This is because that one can always construct a scenario which generates the same workload as accounted by the LHS of (15) (or (16), according to the preemptability of J). This means that if the condition does not hold for a certain combination of the request functions and a specific value of x, then there exists an actual scenario under which J misses its deadline. As a result, the tests provide a necessary condition for schedulability as well.

Require: $v, \tau, B(\tau_{>T}), RF^{\bullet}(\tau_{<T}), RF(\tau_{<T}), RF^{sfx}(v)$
Ensure: v *is schedulable or not*
1: **if** $v \in \mathsf{NPR}(\tau)$ **then**
2: **for all** $\bar{rf} \in RF^{\bullet}(\tau_{<T})$ **do**
3: **for all** $rf^{sfx} \in RF^{sfx}(v)$ **do**
4: **for** $\forall x \leq L$ **do**
5: **if** $\forall t \leq d(v) - e(v) : B(\tau_{>T}) + rf^{sfx}(x) - e(v) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x+t) > x + t$ **then**
6: **return** $false$
7: **else**
8: **for all** $\bar{rf} \in RF(\tau_{<T})$ **do**
9: **for all** $rf^{sfx} \in RF^{sfx}(v)$ **do**
10: **for** $\forall x \leq L$ **do**
11: **if** $\forall t \leq d(v) : B(\tau_{>T}) + rf^{sfx}(x) + \sum_{T' \in \tau_{<T}} rf^{(T')}(x+t) > x + t$ **then**
12: **return** $false$
return $true$

Fig. 3: Algorithm for schedulability of a vertex v

Theorem 1 (Schedulability Test): Conditions (15) and (16) in Lemma 4 provide a necessary and sufficient schedulability test for a job of type v released by task T, for the situations $v \in \mathrm{NPR}(\tau)$ and $v \notin \mathrm{NPR}(\tau)$, respectively.

Figure 3 shows the pseudo-code of the schedulability test. To tackle the combinatorial problem of having to try all combinations of the request functions, we employ the combinatorial abstraction refinement algorithm proposed in [22]. However, details of the abstraction refinement framework is omitted in the pseudo-code.

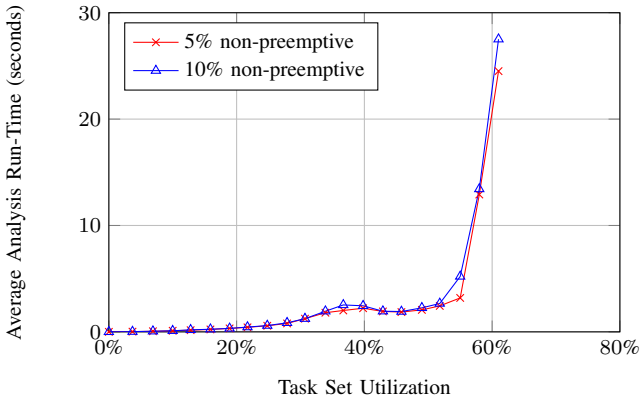
V. Evaluation

A. Experiments with random task sets

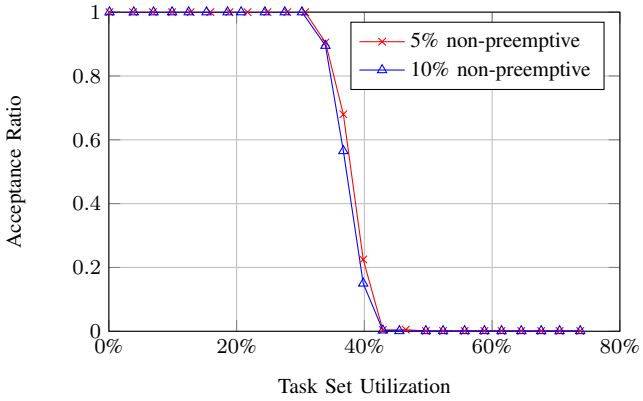
TABLE I: Task set parameters

Job types	Branching degree	p	d/p	e/d
$[3, 5]$	$[1, 3]$	$[50, 200]$	$[0.5, 1]$	$[0, 0.02]$

In this section we evaluate scalability of our proposed method by running it on task sets of different sizes while measuring run-times and acceptance ratios. We use a Python implementation of our methods based on the Python library used in [22]. To generate a task, a random number of job types is created with edges by maintaining a specified range of branching degrees. Minimum inter-release separation times on edge labels are chosen with uniform probability. Then job deadlines are chosen randomly with a uniform ratio to the minimum outgoing edge label and job execution times are chosen randomly with a uniform ratio to the job deadlines. Finally, jobs are randomly marked as non-preemptive according to a given ratio of non-preemptive to total number of jobs. Table I gives the details of the used parameter ranges where p, d and e denote minimum inter-release time, job deadline and job execution time respectively.



(a) Average analysis run-time of the proposed method without timeout for different utilization and ratio of non-preemptive jobs.



(b) Acceptance ratio of the task sets with different utilizations and ratio of non-preemptive jobs.

Fig. 4: Experimental Evaluation

We randomly generate each task set with a given goal of a task set utilization. Here we denote a task utilization as the highest ratio of the sum of job WCETs over the sum of minimum inter-release times in all cycles of a DRT task. Randomly generated tasks are added to a task set until the total utilization of the task set reaches the goal. To see effect of non-preemptive jobs we randomly mark jobs in tasks as non-preemptive to create task sets with 5% and 10% non-preemptive jobs. The motivation of using smaller number of jobs as non-preemptive is that these settings tend to give more feasible task sets.

We evaluate two aspects of our analysis method. One is the analysis run-time for growing task set sizes and the other is the effect of non-preemptive jobs in acceptance ratio. We ran our code in a dual-core processor with frequency 2.10 GHz and a memory of 16 GB. We analyzed about 200 task sets per slot of 3% utilization and measured average run time in each slot. The generated task sets have sizes upto 25 DRT tasks. Although the reported results are platform and task parameter dependent, they provide a qualitative overview of the scalability of our method.

Figure 4a shows average analysis run-time of our method

for task sets with different utilizations. The analysis run-time starts to increase after 30% utilization and increases exponentially after 55% utilization. This is expected, as the busy window extension used in our analysis grows with increasing utilization. A larger busy window also means increased effort to derive request functions in a larger interval. However, the ratio of non-preemptive jobs has little impact on the analysis run-time as it starts to increase in almost identical utilizations for different ratios of non-preemptive jobs. Figure 4b shows that the number of feasible task sets drops to almost zero after 45% utilization. This is still useful considering 4a, as we can analyze potentially feasible task sets before the analysis time starts to increase quickly.

B. Experiments with engine controller

To apply our method in a more realistic scenario, we consider the recently proposed automotive industrial case study by Robert Bosch GmbH [39]. In this case study of Engine Management System (EMS), one of the workload is a engine controller or angle synchronous task. An engine controller task releases different jobs based on different engine speeds. In the original case study, this task is mapped into a separate processor due to its complex release behavior and only one or two very low utilization task is allowed to run with it. However, in a recent work [40], DRT has been shown to be a faithful and fine grained model for such workload. Inspired by this progress, we consider schedulability of a DRT model representing a engine controller (see Figure 5) with randomly generated task set as before. The engine controller DRT task in consideration is fully non-preemptive with worst-case utilization of 23.07%. Additionally we assign the highest task priority to this task to maximize its interference on other co-running tasks. In the experiments, we vary the ratio of non-preemptive jobs from 5% to 20% in randomly generated tasks to see the effect on acceptance ratio. As seen in Table II, acceptance ratio drops very quickly for task sets with more than 10% non-preemptive jobs. On the positive side, we observe that some additional task set with utilization more than 10% is schedulable even in the presence of a highest priority fully non-preemptive engine controller.

Ratio of non-preemptive	Total Utilization	Acceptance ratio
5%	28.7%	96.66%
	32.9%	53.33%
	34.7%	3.03%
10%	28.5%	99.01%
	32.5%	40.04%
	35.3%	0.10%
15%	27,8%	93.33%
	31.9%	31.72%
	35.06%	0.001%
20%	29.1%	86.66%
	31.6%	25.35%
	34.9%	0.00%

TABLE II: Acceptance ratio with engine controller

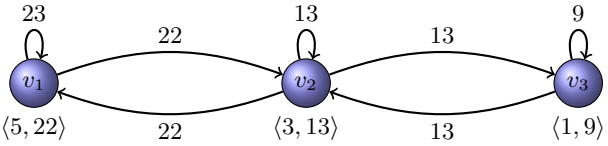


Fig. 5: An engine control DRT task example extracted using the methods of [40]. Job types $v1$, $v2$ and $v3$ represent jobs released in speed range [500,2500), [2500,4500) and [4500,6500) rpm respectively.

VI. Code Generation Approach

In order to implement the semantics of a DRT task set in Ada, we adopt an approach similar to the one proposed in [23]. The main difference between our approach and [23] is in allowing job release by external events and in realizing the non-preemptive executions of a job type. In the following, we first review the time-triggered and event-triggered release of a DRT task implemented in Ada. Then, we discuss the implementation of non-preemptive execution.

```
begin
        Last_Release := Clock;
        loop
            <<T1_loop>>
            case Current_State is
                when s =>
                    s_code;
   -- Task will be blocked here for event
                    Event_receiver.Wait(event_type);
                    if event_type = u then
                        Current_State := u;
                        Last_Release := Clock;
                        goto T1_loop;
                    else if event_type = v then
                        Current_State := v;
                        Last_Release := Clock;
                        goto T1_loop;
                    end if;
                when u =>
 -- Granted access to Event_receiver will raise the
priority
                    Event_receiver.u_code;
 -- Priority will return to the normal one
                    delay until Last_Release + p3;
                    Current_State := w;
                    Last_Release := Last_Release + p3;
                    goto T1_loop;
 -- Respective code is also generated for v and w
                   ...
```

Listing 1: Event-triggered task release

We illustrate our approach using a sample DRT task $T1$ shown in Fig. 6. In this example, we assume that only u is a non-preemptive job; thus, $\mathsf{NPR}(\tau) = \{u\}$. Execution of each job type comprises calling its respective procedure. The code implementing the graph structure, which governs the release times of the jobs, is implemented in the task body, as seen in Listing 1. We assume jobs can be released either time-triggered or event-triggered way. In case of time-triggered job release, for job types (vertices) with more than one outgoing edges, we suppose that each outgoing edge is labeled with a logical condition, called a guard. At runtime, the edge with the minimum inter-release time whose guard evaluates to True is taken.

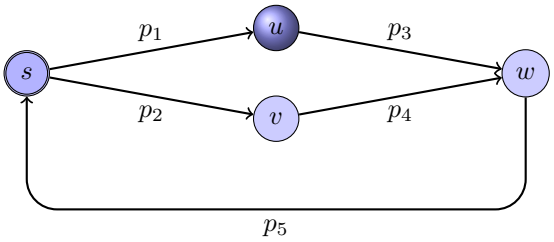


Fig. 6: A sample DRT task T1.

For event-triggered job release, we assume the task to be blocked in a event that is generated by an interrupt and the job releasing events satisfy their corresponding minimum inter-release time between jobs. In our example, Task $T1$ releases job types u and v depending on events that is generated by an external interrupt. The corresponding Ada code is depicted in Listing 1 where $T1$ is blocked on a protected object [27] entry which inherits the priority of the interrupt. As interrupts always have priority higher than any task, execution of any part of that interrupt protected object will be non-preemptive. An example of such a protected object called *Event_receiver* is shown in Listing 2. Notice that the code for non-preemptive job type can be implemented as a procedure of *Event_receiver*. In our current example, the non-preemptive procedure *u_code* of jobtype u is placed in the protected body of *Event_receiver*. As a result, during the execution of *u_code* (Line 21 in Listing 1), the priority of task $T1$ will be raised to the highest system priority and the execution will be non-preemptive. There are three ways to execute a task non-preemptively in Ada. Since Ada 2005, all Ada tasks in a system can execute non-preemptively using Non-Preemptive FIFO Within Priorities task dispatching policy. This policy allows tasks to run to completion unless they are blocked or execute a delay statement. However, this policy does not allow mixed preemptive and non-preemptive execution.

Ada allows changing priority of a task during runtime by calling the library procedure `Set_Priority(Priority)`. A task can raise its priority to the highest system priority before executing the non-preemptive job and return to its original priority after the non-preemptive execution. The problem with this approach is that dynamic priorities are not allowed in Ada restrictions for high integrity systems called the Ravenscar profile [24].

A more general way to execute a non-preemptive job in Ada is to use a protected object with the highest task priority. A protected object is a data structure that encapsulates the shared resource for mutually exclusive access using the Ceiling Locking Protocol (CLP) [26]. Ceiling priorities for protected objects are statically assigned by means of a pragma when the object is created. Protected objects have lock variables, which are inaccessible to the programmer. The compiler includes the necessary operations in the locks. In this case, all the non-preemptive procedures (code blocks) from all the tasks are placed in the protected body of the highest priority protected object. Whenever a task tries to execute its non-preemptive job, it simply calls the respective protected procedure. A successful call raises the priority of the caller task to the ceiling priority during the execution of the procedure. As the protected procedure is executed with the highest priority, all preemptions are disallowed during its execution. This way has two clear advantages over dynamic priorities approach. Firstly, task priority raising/lowering during the non-preemptive execution is handled by the runtime system. Secondly, this type of non-preemptive execution of the protected object is allowed in Ravenscar.

```
1  protected Event_receiver is
2      -- All non-preemptive procedures are declared here
3      entry Wait (Event: event_type ID);
4      -- Highest priority
5      pragma Interrupt_Priority (Interrupt_Priority'Last);
6    procedure u_code;
7    -- Declaration for other non-preemptive procedures
8    ...
9  private
10   procedure Handle_Incoming_Event;
11       pragma Attach_Handler (Handle_Incoming_Event,
     Event_Arrival);
12       Buffer : Contents (Message_Size);
13       Ready : Boolean := False;
14 end Event_receiver;
15
16 protected body Event_receiver is
17
18       procedure Handle_Incoming_Event is
19       begin
20           -- interrupt handler code
21           Ready := True;
22       end Handle_Incoming_Event;
23
24       entry Wait (Event: event_type ID) when Ready is
25       begin
26          -- this entry will be unblocked by the interrupt
27          Ready := False;
28       end Wait;
29
30       procedure u_code is
31       begin
32      -- This code will be executed non-preemptively
33   end u_code;
34
35       -- Defintion of the other non-preemptive procdures
     as part of this protected object body
36   ...
37 end Event_receiver;
```

Listing 2: Implementation of a non-preemptive job using interrupt handling protected object.

As we see from the above discussion, the protected object approach is the most suitable way to implement the mixed preemptive non-preemptive execution of jobs in Ada.

VII. Conclusion

DRT task model can precisely capture state-dependent workload via multiple job types. However to be considered for multi-task implementation, DRT should be extended with resource sharing. One effective way to achieve this is to non-preemptively execute the jobs that use shared resource. In this paper, we provide an exact schedulability test method for DRT tasks with mixed preemptive non-preemptive job execution. Experimental evaluation shows the efficiency of the method to analyze potentially feasible task sets. Meanwhile, we propose an Ada implementation which preserves the event-triggered semantics of DRT tasks and achieves non-preemptive execution by employing protected objects.

We identify following future works based on the results of this paper:

- A line of future research is to explore resource sharing protocols such as the Priority Ceiling Protocol (PCP) and the Priority Inheritance Protocol (PIP) [42] to understand their overhead and suitability for DRT tasks.
- An interesting extension of the proposed method is to consider non-preemeptive resource sharing in case of Fork-Join DRT task [20] which is more suitable for multiprocessor platforms.
- A tool implementation incorporating the proposed timing analysis methods with graphical modeling facilities would be useful to explore more realistic case studies.

REFERENCES

[1] Simulink User's Guide. The MathWorks, Inc. Available: https://www.mathworks.com/help/pdf_doc/simulink/sl_using.pdf, 2016.

[2] M. Stigge, P. Ekberg, N. Guan and W. Yi, The Digraph Real-Time Task Model, in Proc. RTAS, pp.71–80, 2011.

[3] M. Stigge and W. Yi, Graph-based models for real-time workload: a survey, in J. Real-Time Syst. 51(5), 602–636, 2015.

[4] G. C. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications (3rd ed.). Springer Publishing Company, Inc. 2003.

[5] A. Mok and A. Chen, A multiframe model for real-time tasks, in Proc. RTSS, pp. 22–29, 1996.

[6] G. C. Buttazzo, M. Bertogna and G. Yao, Limited Preemptive Scheduling for Real-Time Systems. A Survey, in IEEE Trans. Ind. Informat. 9(1), 3–15, 2013.

[7] A. Baldovin, E. Mezzetti and T. Vardanega, Limited Preemptive Scheduling of Non-independent Task Sets, in Proc. EMSOFT, pp.18:1-18:10, 2013.

[8] M. A. Serrano, A. Melani, M. Bertogna and E. Quinones, Response-time analysis of DAG tasks under fixed priority scheduling with limited preemptions, in Proc. DATE, pp. 1066-1071, 2016.

[9] K. Tindell and A. J. Wellings, An extendible approach for analyzing fixed priority hard real-time tasks, in Real-Time Syst. 6(2), 133–151, 1994.

[10] K. Jeffay, Scheduling sporadic tasks with shared resources in hard-real-time systems, in Proc. RTSS, pp. 89–99, 1992.

[11] P. Ekberg, N. Guan, M. Stigge and W. Yi, An optimal resource sharing protocol for generalized multiframe tasks, in Journal of Logical and Algebraic Methods in Programming. 84(1), 92–105, 2011.

[12] J. P. Lehoczky, Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines, in Proc. RTSS, pp. 201–209, 1990.

[13] R. J. Bril, J. J. Lukkien and W. F. Verhaegh, Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited, in Proc. ECRTS, pp. 201–209, 2007.

[14] J. Rox and R. Ernst, Exploiting Inter-Event Stream Correlations Between Output Event Streams of non-preemptively Scheduled Tasks, in Proc. DATE, 2010.

[15] J. Maki-Turja and M. Nolin, Efficient implmentation of tight response-times for tasks with offsets, in Real-Time Systems, 40 (1), pp. 77–116, 2008.

[16] P. S. Kurtin, J. P. H. M. Hausmans and M. J. G. Bekooij, Combining offsets with precedence constraints to improve temporal analysis of cyclic real-time streaming applications, in Proc. RTAS, 2016.

[17] R. I. Davis, A. Burns, R. J. Bril and J. J. Lukkien, Controller area network (CAN) schedulability analysis. Refuted, revisited and revised, in Real-Time Systems, 35(3), 239–272, 2007.

[18] L. George, N. Rivierre and M. Spuri, Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling, [Research Report] RR-2966, INRIA. 1996.

[19] S. K. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, L. Stougie and A. Wiese, A generalized parallel task model for recurrent real-time processes, in Proc. RTSS, pages 6372, 2012.

[20] M. Stigge, P. Ekberg and W. Yi, The fork-join real-time task model, SIGBED Rev. 10, 2 (July 2013), 20-20, 2013.

[21] M. Stigge and W. Yi, Combinatorial Abstraction Refinement for Feasibility Analysis. in Proc. RTSS, pp.340–349, 2013.

[22] M. Stigge and W. Yi, Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities. in J. Real-Time Syst. 51(6), 639–674, 2015.

[23] J. Abdullah, M. Mohaqeqi and W. Yi, Synthesis of Ada Code from Graph-Based Task Models, in Proc. SAC, pp. 1466–1471, 2017.

[24] A. Burns, B. Dobbing and T. Vardanega, Guide for the use of the Ada Ravenscar Profile in high integrity systems, in Ada Lett. XXIV, 1–74, 2004.

[25] Stateflow Documentation. The MathWorks, Inc. Available: https://www.mathworks.com/help/stateflow/, 2017.

[26] M. H. Klein, T. Ralya, B. Pollak, R. Obenza and M. G. Harbour, A practitioner's handbook for real-time analysis. Kluwer Academic Publishers, Norwell, MA, USA. 1993.

[27] A. J. Wellings, B. Johnson, B. Sanden, J. Kienzle, T. Wolf and S. Michell, Integrating object-oriented programming and protected objects in Ada 95, in TOPLAS, 22(3), pp. 506–539, 2000.

[28] T. Henzinger, B. Horowitz, C. Kirsch, Giotto: A time-triggered language for embedded programming, in Proceedings of the IEEE, 91(1), 84–99, 2003.

[29] S. Resmerita, W. Pree, Verification of embedded control systems by simulation and program execution control, in American Control Conference, 3581–3586, 2012.

[30] A. Cervin, B. Lincoln, J. Eker, K. rzen, How Does Control Timing Affect Performance? Analysis and Simulation of Timing Using Jitterbug and TrueTime, in IEEE Control Systems Magazine, 23(3), 16–30, 2003.

[31] J. Liu and E. Lee, Timed multitasking for real-time embedded software, in IEEE Control Systems Magazine, 23:65-75, 2002.

[32] S. Baruah, Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In: Real-Time and Network Systems (RTNS), 11–19, 2012.

[33] S. Edwards, E. Lee, The semantics and execution of a synchronous block-diagram language, in Sci. Comp. Prog., 48:2142(22), 2003.

[34] P. Caspi, N. Scaife, C. Sofronis and S. Tripakis, Semantics-preserving multitask implmentation of synchronous programs, in ACM Transactions on Embedded Computing Systems (TECS), 7(2), 15:1–15:40, 2008.

[35] Simulink Embedded Coder. The MathWorks, Inc. Available: https://www.mathworks.com/products/embedded-coder/, 2016.

[36] KCG Code Generator. ANSYS Inc. Available: http://www.esterel-technologies.com/products/scade-suite/, 2016.

[37] MAST toolset, Available: http://mast.unican.es/.

[38] UML MARTE profile, Available: http://www.omg.org/spec/MARTE/.

[39] 2017 Formal Methods for Timing Verification (FMTV) challenge, colocated with the 8th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS). https://waters2017.inria.fr/challenge/

[40] M. Mohaqeqi, J. Abdullah, P. Ekberg and W. Yi, Refinement of Workload Models for Engine Controllers by State Space Partitioning, in Proc. ECRTS, 11:1-11:22, 2017.

[41] M. Di Natale, L. Guo and H. Zeng, A. Sangiovanni-Vincentelli, Synthesis of Multitask Implementations of Simulink Models With Minimum Delays, in IEEE Trans. Ind. Informat. 6(4), 637–651, 2010.

[42] L. Sha, R. Rajkumar, J. P. Lehoczky, Priority inheritance protcols: An approach to real-time synchronization, in IEEE Tranasction on Computers. 39(9), 1175–1185, 1990.

[43] P. Deng, Q. Zhu, M. Di Natale and H. Zeng, Task synthesis for latency-sensitive synchronous block diagram. In: Proc. of SIES, 112–121, 2014.