Schedulability Analysis of Synchronous Digraph Real-Time Tasks

Morteza Mohaqeqi*, Jakaria Abdullah*, Nan Guan† and Wang Yi*
*Uppsala University, Sweden
†Northeastern University, China

Authors' Version

***Abstract*—Real-time task models have evolved from periodic models to more sophisticated graph-based ones like the Digraph Real Time task model (DRT) to specify branching and loop structures of real-time embedded software. For independent DRT tasks, efficient techniques for schedulability analysis have been developed in previous work. In this paper, we extend the DRT model to specify inter-task synchronization through a rendezvous mechanism. We present an abstraction technique for static priority schedulability analysis of the corresponding tasks. Our experiments show that, despite the high computational complexity of the problem, the proposed technique scales very well for large sets of dependent tasks.**

I. Introduction

With the increased complexity of real-time embedded software, more expressive task models are required to characterize the resource requirements of the real-time tasks. In the past, a number of task models, ranging from the relatively simple periodic and sporadic ones to the more complex graph-based ones, have been proposed. The Digraph Real-Time (DRT) model [1] is known as one of the most expressive ones which can be used for specifying complex structures of real-time programs such as branching and loop structures. Based on this model, a task may consist of different job types, where the dependencies among the jobs are specified by a directed cyclic graph.

While efficient schedulability analysis methods have been developed for the DRT task model [1], [2], [3], previous work has been mainly based on the tasks independence assumption [4]. More specifically, while the job dependency inside a task can be specified by the model, no explicit and general notion of inter-task dependency has been specified. The schedulability analysis of task systems with inter-task dependency is often more sophisticated, which makes the existing techniques unusable for this case. For instance, regarding the fixed priority scheduling policy, a simplifying property is that lower priority tasks have no influence on the execution of higher priority ones. However, in a system with task dependency, execution of a high priority task can be influenced by the jobs belonging to a lower priority one.

In this paper, we present a new task model for specifying inter-task synchronization of real-time tasks through a rendezvous mechanism. We develope techniques to perform uniprocessor schedulability analysis of this model. For this purpose, we extend the well-known notion of *request function* [2] by augmenting it with the additional information of the synchronization instants. Additionally, two abstraction refinement techniques are proposed for improving the analysis efficiency. The experimental results show that despite the essential complexity of the problem, the method scales very well for large and complex task sets.

The rest of this paper is organized as follows. Section II reviews related work. The task model is formally defined in Section III. Our approach for static priority schedulability analysis of the proposed task model is presented in Section IV. A number of techniques for improving the analysis method are provided in Section V. Section VI evaluates the efficiency and scalability of the analysis method.

II. Related Work

Much work has been done on the schedulability analysis of various graph-based real-time task models, such as the multi-frame (MF) task model [5], generalized multiframe (GMF) task model [4], non-cyclic GMF task model [6], recurring branching (RB) task model [7], recurring real-time (RRT) task model [8], etc. Figure 1 shows the generalization relations among these task models. As it is seen, the DRT task model considered in this paper is a generalization of the above models. In the past, efficient analysis methods have been proposed for both dynamic-priority [1] and static-priority [2] scheduling analysis of DRT tasks. Further, Guan et al. [3] proposed two approximate response-time analysis methods and derived the respective speed-up factors. These methods, however, are unable to model the dependencies between real-time tasks.

The synchronization semantic (called rendezvous or synchronous message passing) considered in this paper is widely used in modeling formalizations (such as CSP [9], Petri net [10]), programming languages (such as Ada [11], which is popular in developing real-time systems, Occam [12], SHIM [13]) and real-time systems modeling tools (such as UPPAAL [16], Ptolemy II [14]). Rendezvous is closely related to the barrier synchronization that is widely used in parallel programming models (e.g., the fork-join model [15]).

In the context of formal verification, rendezvous synchronization is supported by model checking tools such as UP-

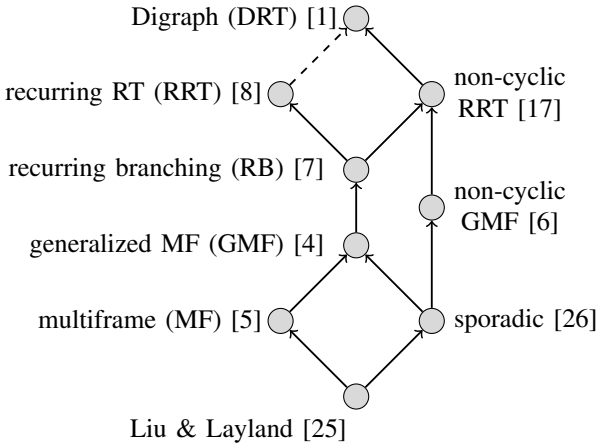


Fig. 1: Generalization relations between task models, taken from [3].

PAAL [16] and TIMES [18]. Due to the analysis complexity of automata, the above mentioned tools suffer from the state space-explosion problem. Rendezvous has been also considered in the context of periodic real-time task models in early works [19], [20], [21]. However, these scheduling and analysis techniques are not applicable to the DRT model of this paper due to the combinatorial state space explosion caused by the graph structures.

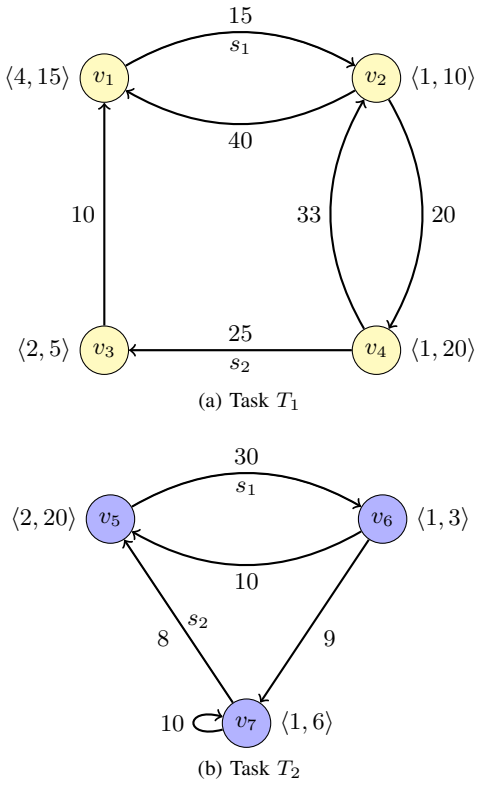
Limited studies have been done on synchronization of DRT tasks. Guan et al. [22] develop a resource sharing protocol to deal with branching structures in DRT tasks, giving a speedup factor of 1.618 when used with uniprocessor fixed priority scheduling. In addition, a synchronization model has been specified in [23] in which jobs of different DRT tasks can be synchronized. Compared to the mentioned studies [22], [23], in this paper we consider a different kind of synchronization where releases of jobs from different tasks are synchronized on action labels denoted over their inter-release edges. As a special case of synchronization, the fork-join DRT task model has been specified by Stigge et al. [24]. In that model, a DRT task can take a *fork* edge. When a fork edge is taken, a set of independent paths will be followed in parallel until they are joined in a so-called *join* edge. In contrast to our work, they studied the EDF schedulability of this model.

III. Task Model

This section specifies the syntax and semantics of the synchronous digraph real-time task model. In addition, notations used in the subsequent sections are introduced.

A. Syntax

A synchronous digraph real-time (SDRT) task T is specified by a directed graph $G(T) = (V(T), E(T))$, where $V(T)$ and $E(T)$ denote the set of graph's vertices and edges, respectively. Each vertex $v \in V(T)$ represents a *job type*, and is labeled with a pair $\langle e(v), d(v)\rangle$, where $e(v)$ denotes the worst-case execution time (WCET) and $d(v)$ denotes the relative deadline of the corresponding jobs. It is supposed that WCETs and relative deadlines are positive integers. Each edge $(u, v) \in E(T)$ is also labeled with a positive integer, $p(u, v)$, specifying the minimum inter-release time between two jobs. In addition, an edge (u, v) may be labeled with an action which is denoted by $a(u, v)$. Actions are used to denote inter-task synchronization. We use the notation $a(u, v) = \perp$ to show that an edge (u, v) is not associated with any synchronization.



(a) Task T_1

(b) Task T_2

Fig. 2: Two SDRT tasks with two synchronizations on actions s_1 and s_2.

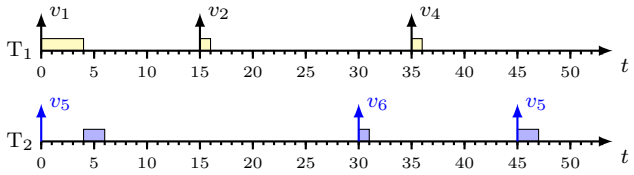
Two tasks T_1 and T_2 are said to have a synchronization on action s if there exist edges $(u, v) \in E(T_1)$ and $(u', v') \in E(T_2)$ such that $a(u, v) = s$ and $a(u', v') = s$.[1] Moreover, the set of synchronization actions between the two tasks is represented as $Act(T_1, T_2)$. More precisely, $Act(T_1, T_2)$ is defined as

$$Act(T_1, T_2) = \{s \mid a(u, v) = a(u', v') = s \neq \perp \text{ for some } (u, v) \in E(T_1) \text{ and } (u', v') \in E(T_2)\}$$

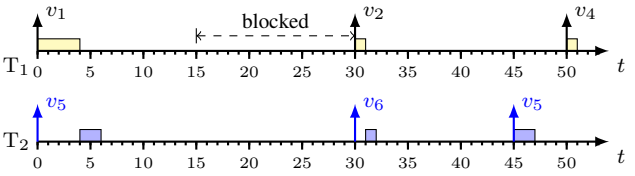
Example 1: Figure 2 shows two SDRT tasks which have two synchronizations on actions s_1 and s_2. For these tasks, we have $Act(T_1, T_2) = \{s_1, s_2\}$.

In this paper, our focus is on the SDRT tasks with constrained deadline. This means that, given an SDRT task T, for each $u \in V(T)$, we have $d(u) \leq p(u, v)$ for all $(u, v) \in E(T)$. Also, similar to the DRT task model, it is assumed that for any $u, v \in V(T)$, there exists at most one edge from u to v.

[1] In this paper, we assume that exactly two tasks are involved in each synchronization (action).



(a) Asynchronous release



(b) Synchronous release

Fig. 3: Sample executions of the tasks specified in Fig. 2.

B. Semantics

The semantics of the SDRT task model is defined based on the set of execution traces which can be generated by the respective tasks. An execution trace of a task is specified by a *job sequence*.

Definition 1 (Job Sequence [1]): A job sequence is defined as $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$, where each tuple (R_i, e_i, v_i) represents a job instance, in which R_i, e_i, and v_i denote job release time, job execution time, and the job type, respectively. Additionally, the job sequence σ can be generated by a task T if $(v_0, v_1, \ldots)$ is a path in the task's graph $G(T)$, and

- $R_{i+1} - R_i \geq p(v_i, v_{i+1})$, and
- $e_i \leq e(v_i)$,

for all $i \geq 0$. A job sequence may be finite or infinite.

An SDRT task can only generate *synchronous* job sequences. In a synchronous execution, the jobs of two tasks which are associated with a common synchronization action must be released at the same time. If one of the synchronized jobs is ready to be released while the other one is not, it will be blocked until the other job becomes ready. This is similar to the synchronization (rendezvous) mechanism provided by the modeling tools (such as UPPAAL [16]) and programming languages (such as Ada [27]) through which the communication and synchronization of the tasks can be specified.

Figure 3 shows sample executions of the tasks T_1 and T_2 specified in Fig. 2. A non-synchronous execution (which is not valid according to the synchronization semantic) is depicted in Fig. 3a. Besides, Fig. 3b illustrates a synchronous execution in which task T_1 is blocked until the other task (namely T_2) can release the respective job, v_6.

In order to formally specify the synchronization semantics, we first define the notion of *action sequence*.

Definition 2 (Action Sequence): The action sequence (AS) of a job sequence $\sigma = [(R_0, e_0, v_0), \ldots, (R_n, e_n, v_n)]$ is defined as

$$AS_\sigma \coloneqq [(s_0, t_{s_0}), \ldots, (s_m, t_{s_m})] \tag{1}$$

where a pair (s_j, t_{s_j}) is in AS_σ if and only if there exists some i, $0 \leq i < n$, for which $s_j = a(v_i, v_{i+1}) \neq \perp$ and $t_{s_j} = R_{i+1}$.

It is supposed that an action sequence is sorted with respect to the release times, that is, $t_{s_j} < t_{s_{j+1}}$, for $0 \leq j < m$. Also, for a set of actions $\mathcal{A}$, let $\lfloor AS_\sigma \rfloor_\mathcal{A}$ denote the action sequence obtained from AS_σ by removing all the tuples (s, t_s) for which $s \notin \mathcal{A}$. On the basis of these definitions, we define the notion of synchronous job sequence.

Definition 3 (Synchronous Job Sequences): Two job sequences σ and σ' generated by two arbitrary SDRT tasks T_1 and T_2 are said to be synchronous if $\lfloor AS_\sigma \rfloor_{Act(T_1,T_2)} = \lfloor AS_{\sigma'} \rfloor_{Act(T_1,T_2)}$. Further, n (for $n > 2$) job sequences are synchronous if any two of them are synchronous.

Example 2: Consider the SDRT tasks shown in Fig. 2. Two job sequences $\sigma = [(0, 4, v_1), (15, 1, v_2), (35, 1, v_4)]$ and $\sigma' = [(0, 2, v_5), (30, 1, v_6), (39, 1, v_7)]$, generated by T_1 and T_2, are not synchronous and thus are not regarded as valid job sequences. Meanwhile, the job sequence $\sigma'' = [(0, 4, v_1), (30, 1, v_2), (50, 1, v_4)]$ generated by T_1 is synchronous with σ' since $\lfloor AS_{\sigma''} \rfloor_{\{s_1,s_2\}} = \lfloor AS_{\sigma'} \rfloor_{\{s_1,s_2\}} = [(s_1, 30)]$.

C. Further Definitions

In this section, further definitions and notations which are used in the subsequent sections are introduced.

Consider an SDRT task and an arbitrary path $\pi = (v_0, v_1, \ldots, v_l)$ in the respective graph. Then, the *most dense* job sequence generated via π is defined as $\sigma_\pi = [(R_0, e_0, v_0), \ldots, (R_l, e_l, v_l)]$, where

- $R_0 = 0$,
- $R_i = \sum_{j=0}^{i-1} p(v_j, v_{j+1})$, for $0 < i \leq l$, and
- $e_i = e(v_i)$, for $0 \leq i \leq l$.

While the most dense job sequence of a path is unique, infinite number of job sequences can be associated with each path. To show this, we define the notion shifted job sequence as follows.

Definition 4 (Job Sequence Shifting): Assume that $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ is a job sequence generated by a task T. In addition, let i and t be two arbitrary positive integers. Then, the *shifted* job sequence with respect to i and t is defined as the job sequence $\sigma'(i, t) = [(R'_0, e_0, v_0), (R'_1, e_1, v_1), \ldots]$ where

$$R'_j = \begin{cases} R_j, & j < i, \\ R_j + t, & j \geq i, \end{cases}$$

Lemma 1: For any job sequence σ generated by a task T, all of the corresponding shifted job sequences can also be generated by T.

Proof: It is easily seen that the shifted job sequence satisfies the conditions specified in Definition 1. As a result, it is also a job sequence. ■

It is worth noting that if σ' is a job sequence obtained by shifting a job sequence σ, then σ and σ' are generated through the same path in the respective task graph.

For an action sequence $AS = [(s_0, t_{s_0}), \ldots, (s_m, t_{s_m})]$, we use the notation $AS[i]$ to refer to the $i+1$th tuple, namely (s_i, t_{s_i}). Also, we define $AS[i].s \equiv s_i$ and $AS[i].t \equiv t_{s_i}$. Further, $|AS|$ denotes the size of AS.

For convenient presentation and without loss of generality, it is assumed that multiple actions of the same type appearing in a job sequence are relabeled to different and unique labels. For example, two instances of an action s appearing in a job sequence can be relabeled as y and z, and are dealt with as different actions. This relabeling is accomplished such that if the actions of two job sequences match with each other, the respective actions after the relabeling will match as well.

IV. Schedulability Analysis

In this section, we present schedulability analysis of the SDRT task sets for the static priority (SP) scheduling policy. Under an SP scheduling policy, a unique priority is assigned to each task. At runtime, a job of a task can be executed only if no job of higher priority tasks exists in the system. Accordingly, SP schedulability of a task set is defined as follows.

Definition 5: ([2]) A task set is said to be SP schedulable if and only if there exists at least one priority order for the tasks under which all the jobs meet their deadline.

For schedulability analysis, we first review the SP schedulability conditions for a set of DRT tasks [2]. Afterwards, we present our analysis approach for the SDRT task model.

A. SP Schedulability for DRT Tasks

A DRT task is specified by a directed graph as done in the SDRT task model. The difference is that no synchronization can be specified by the DRT task model. The DRT SP schedulability analysis approach proposed in [2] is based on the notion of request function. Intuitively, a request function determines the maximum accumulated workload which can arrive during a time interval.

Definition 6 (Request Function [2]): Consider an arbitrary path $\pi = (v_0, v_1, \ldots, v_l)$ in a task graph and the respective most dense job sequence $\sigma_\pi = [(R_0, e_0, v_0), \ldots, (R_l, e_l, v_l)]$. The request function associated to π is defined as

$$rf_\pi(t) := \max_k \left\{ \sum_{i=0}^{k} e_i \;\middle|\; R_k < t \right\} \tag{2}$$

for $t > 0$, and $rf_\pi(t) = 0$ for $t \leq 0$.

The SP schedulability of a given job with respect to a set of higher priority tasks (called interfering tasks) can be specified based on this definition. To this aim, define Π_T as the set of all paths in the graph of task T. Also, for a set of tasks $\tau = \{T_1, T_2, \ldots, T_n\}$, let $\Pi(\tau) = \Pi_{T_1} \times \Pi_{T_2} \times \ldots \times \Pi_{T_n}$ be the set of all path combinations, namely

$$\Pi(\tau) = \{(\pi_1, \ldots, \pi_n) \mid \pi_1 \in \Pi_{T_1}, \ldots, \pi_n \in \Pi_{T_n}\}$$

Then, the SP schedulability condition of a job belonging to a DRT task with respect to a set of interfering DRT tasks is given by the following theorem.

Theorem 1 ([2]): A job with worst-case execution time e and relative deadline d is schedulable, i.e., meets its deadline, under a set of interfering (i.e., higher priority) tasks τ if and only if

$$\forall(\pi_1, \ldots, \pi_n) \in \Pi(\tau) : \exists t \leq d : e + \sum_{T_i \in \tau} rf_{\pi_i}(t) \leq t \tag{3}$$

In other words, a necessary and sufficient condition for schedulability of a job is that there exists a time instant t between the job release time and its deadline at which the job and all of the higher priority jobs released before t are completely executed.

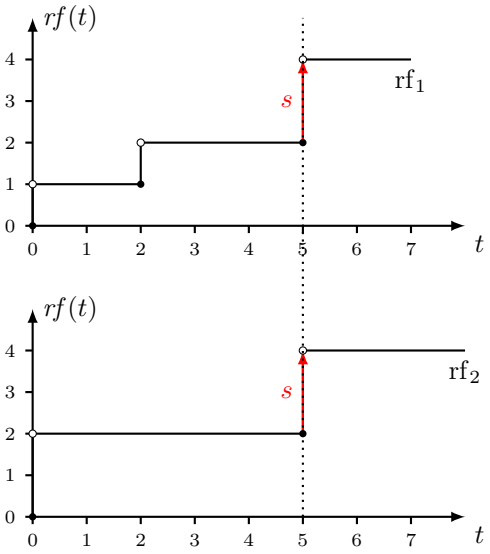


Fig. 4: Two synchronous request functions.

B. SP Schedulability for SDRT Tasks

According to the specified semantics for the SDRT task model, a synchronization can influence the timing characteristics and resource requests of the tasks. This reveals that one cannot directly apply Theorem 1 to the case of SDRT. In this section, we first adapt the definition of request function such that the information of synchronization instants can be represented as well. To this aim, in addition to the accumulated arrived workload, an *action sequence* is also associated with a request function.

The action sequence of a request function rf, denoted as AS_{rf}, is defined as the action sequence of the corresponding job sequence as defined in Definition 2. We use this definition to specify *synchronous* request functions. Two request functions are said to be synchronous if they contain the same action sequences (considering only the actions that are common between the respective tasks). Further, a set of (more than two) request functions are synchronous if any two of them are synchronous. Figure 4 shows an example of two synchronous request functions, where the synchronization instant is denoted by an upward arrow.

According to the synchronous semantics of the SDRT task model, for SP schedulability analysis (as defined in Definition 5) we need to only consider synchronous request

functions. A brute force approach is to generate all paths and the corresponding job sequences, and then, to select the synchronous ones to apply the SP schedulability condition. This approach, however, is quite inefficient since a large number of *useless* non-synchronous job sequences are generated and checked. To address this issue, we propose an approach which only considers the synchronous job sequences. In summary, we first generate all of the *most dense* job sequences for each task. Then, for any combination of the corresponding request functions from different tasks, an *alignment* is performed to obtain the synchronous counterparts. After that, the schedulability condition stated in Theorem 1 is applied. We show that this approach is sound in terms of determining the SP schedulability of an SDRT task set as defined in Definition 5. The details are provided in the following.

We first define the notion of *extended* action sequences of a given job sequence. For this purpose, consider a job sequence $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ generated by task T with the corresponding action sequence AS_σ. We define the set of extended action sequences as

$$EAS_\sigma = \{AS_\sigma\} \cup \big\{[(s,0), AS_\sigma[0], AS_\sigma[1], \ldots] \mid \exists (u, v_0) \in E(T) : a(u, v_0) = s \ \big\}$$

Accordingly, for a path π and its most dense job sequence σ_π, RF_π is defined as the set of all request functions rf, for which $rf(t)$ is calculated as in (2), and $AS_{rf} \in EAS_{\sigma_\pi}$.

As mentioned, for SP schedulability, we need to consider the synchronous request functions associated to a task set. However, the request functions obtained from the most dense job sequences are not necessarily synchronous. We introduce the alignment operation to obtain the corresponding synchronous request functions.

Definition 7 (Request Function Alignment): Consider two request functions rf and rf' where a common synchronization action s exists in their action sequences. Additionally, let (s_i, t_{s_i}) and (s_j, t_{s_j}) denote the respective tuples in AS_{rf} and $AS_{rf'}$, respectively (i. e., $s_i = s_j = s$). Without loss of generality, we assume $t_{s_i} \leq t_{s_j}$. The alignment operation of rf and rf' with respect to s is denoted as $align(rf, rf', s)$ and is defined as follows. Let arf and arf' be the resulting request functions associated to rf and rf', respectively, after the alignment. We define $arf' = rf'$ and

$$arf(t) = \begin{cases} rf(t), & t \leq t_{s_i}, \\ rf(t_{s_i}), & t_{s_i} < t \leq t_{s_j}, \\ rf(t - \delta_s), & t > t_{s_j}, \end{cases} \tag{4}$$

where $\delta_s = (t_{s_j} - t_{s_i})$. Additionally, the respective action sequences are specified as $AS_{arf'} = AS_{rf'}$ and

$$AS_{arf} = [AS_{rf}[0], \ldots, AS_{rf}[i-1], AS_{rf}[i] + \delta_s, AS_{rf}[i+1] + \delta_s, \ldots],$$

where $AS_{rf}[i] + \delta_s \equiv (AS_{rf}[i].s, AS_{rf}[i].t + \delta_s)$.

Based on this definition, we define the *Synch* operation for a set of request functions as shown in Fig. 5. As seen, first, the function *Align* is called, which finds the pair of request functions whose first actions are the same (lines 1 and 6 in Fig. 6). If found, the request functions are aligned (line 4 in Fig. 6) and the procedure proceeds to the next actions. Otherwise, the request functions cannot be aligned anymore. Thus, the arriving workload after the first unmatched synchronization must be ignored (lines 2 to 9 of Fig. 5) since the synchronization cannot take place, which means that the respective task is blocked.

The *Synch* operation defined in Fig. 5 has this property that the resultant request functions constitute a set of synchronous request functions.

Lemma 2: Consider a set of tasks $\tau = \{T_1, ..., T_n\}$. Let $\bar{\pi} = (\pi_1, ..., \pi_n) \in \Pi(\tau)$ be an arbitrary path combination where $\pi_i \in \Pi_{T_i}$, and define

$$RF_{\bar{\pi}} = \{\{rf_1, \ldots, rf_n\} \mid rf_1 \in RF_{\pi_1}, \ldots, rf_n \in RF_{\pi_n}\} \tag{5}$$

For any $R \in RF_{\bar{\pi}}$, the alignment of R as defined in Fig. 5 yields a set of request functions that are related to at least one set of synchronous job sequences generated by τ.

Proof: According to Definition 7, the $align(.)$ operation just shifts (postpones) the arrived workload. Therefore, it can be thought of as the *shift* operation for the job sequences specified in Definition 4. Hence, based on Lemma 1, there exists a valid job sequence generated by the respective task for each aligned request function. Moreover, it can be observed that when two request functions are aligned with respect to an action s (line 4 in Fig. 6), the value of the request

function $Synch(R)$ ▷ R is a set of request functions
1: $Align(R)$ ▷ defined in Fig. 6
2: **for** each $rf \in R$ **do**
3: **if** $|AS_{rf}| > 0$ **then**
4: $(s, t_s) \leftarrow AS_{rf}[0]$
5: $new\_rf(t) = \begin{cases} rf(t), & \text{if } t < t_s, \\ rf(t_s), & \text{otherwise,} \end{cases}$
6: $rf \leftarrow new\_rf$
7: $AS_{rf} \leftarrow \{\}$
8: **end if**
9: **end for**

Fig. 5: Procedure for synchronizing the set R containing one request function per task.

function $Align(R)$ ▷ R is a set of request functions
1: $S \leftarrow \big\{(rf, rf') \mid rf, rf' \in R, AS_{rf}[0].s = AS_{rf'}[0].s\big\}$
2: **while** S is not empty **do**
3: **for** each $(rf, rf') \in S$ **do**
4: $Align\_and\_Pop(rf, rf', AS_{rf}[0].s)$ ▷ see Fig. 7
5: **end for**
6: $S \leftarrow \big\{(rf, rf') \mid rf, rf' \in R,$
7: $AS_{rf}[0].s = AS_{rf'}[0].s\}$
8: **end while**

Fig. 6: Procedure for aligning request functions with respect to their common actions.

function $Align\_and\_Pop(rf, rf', s)$
1: $align(rf, rf', s)$ ▷ defined in Definition 7
2: $AS_{rf} \leftarrow AS_{rf} \backslash AS_{rf}[0]$ ▷ remove $AS_{rf}[0]$
3: $AS_{rf'} \leftarrow AS_{rf'} \backslash AS_{rf'}[0]$ ▷ remove $AS_{rf'}[0]$

Fig. 7: Procedure for aligning two request functions and then removing the first actions.

functions, as well as the tuples in their action sets, that are related to time instants $t \leq t_s$ will not change later in the procedure. Additionally, in the *for* loop in Fig. 5, the actions that are not matched (synchronized) are removed from the request functions. As a result, the obtained request functions are synchronous. ∎

Consequently, we can provide the SDRT SP schedulability condition. Here, we focus on the schedulability of a job which is released independent of other jobs, i.e. a job which has no synchronization with any job of the other tasks. In Appendix A we have shown that how the analysis approach is extended to the general case.

Theorem 2: Given an SDRT task set τ, a job v with worst-case execution time e and relative deadline d is schedulable under an interfering task set $\tau_{hp} \subseteq \tau$ if and only if

$$\forall \bar{\pi} \in \Pi(\tau), \forall R \in RF_{\bar{\pi}} : \exists t \leq d : \quad e + \sum_{\substack{rf_i \in Synch(R) \\ T_i \in \tau_{hp}}} rf_i(t) \leq t, \quad (6)$$

where $RF_{\bar{\pi}}$ is as defined in (5).[2]

Proof: First, we assume Condition (6) holds but the job v is unschedulable. Unschedulability of v means that there must be a combination of request functions $rf_i \in Synch(R)$ which can generate a workload such that for each $t \leq d$, tasks from τ_{hp} are executing strictly more than $t - e$ time units within $[0, t]$. As a result, the sum of $rf_i \in Align(R)$ for tasks in τ_{hp} is strictly greater than $t - e$ for each $t \leq d$, which contradicts Condition (6).

Next we assume that v is schedulable but Condition (6) does not hold. Since v is schedulable, there exists a minimal $t_0 \leq d$ such that e time units of idle time is available in $[0, t_0]$ for any combination of $rf_i(t_0) \in Align(R)$ belonging to the tasks of τ_{hp}. That means any accumulated workload released by the high priority tasks does not exceed $t_0 - e$ within $[0, t_0]$, which is precisely equivalent to Condition (6) leading to a contradiction. ∎

V. State-Space Reduction

The schedulability analysis problem of the DRT task model has been shown to be strongly coNP-hard [28] for SP scheduling. Since the SDRT task model is a generalization of DRT, the exponential complexity of the analysis problem for the SDRT is unavoidable. In this section, we present techniques which, in spite of this complexity, significantly improve the efficiency of the analysis method and make it quite practical to be used for large task sets.

The schedulability analysis method proposed in the previous section explores all path combinations, which leads to an exponential complexity. Mainly, there are two sources causing this complexity. First, the number of paths generated by each task graph can grow exponentially. Second, the number of path combinations obtained from combining paths of different tasks may result in a huge state-space. In this section, we deal with these issues.

For the first concern, it is seen that we do not need to consider all paths (and request functions) generated by an SDRT task. In fact, we define a notion of *dominance* for request functions in the sense that only dominant request functions should to be regarded in the schedulability analysis. Besides, in order to avoid the investigation of all combination of paths, we employ an abstraction approach [2], based on which the set of request functions are first aggregated to abstract request functions. Using this abstraction, unnecessary path combinations can be avoided. Additionally, we provide an abstraction refinement approach based on abstracting the task set with ignoring a subset of synchronization actions. In the following subsections, the details of each approach are specified.

A. Dominant Request Functions

In the SP schedulability analysis, it is observed that a request function may dominate another one such that schedulability under the dominant request function guarantees the schedulability under the dominated one. In such a situation, it suffices to consider only the dominant one in the schedulability analysis (i.e. Condition (6)). Definition 8 formally specifies this dominance relation.

Definition 8: Consider two request functions rf_1 and rf_2. rf_1 dominates rf_2 if it holds that:

If Eq. (6) holds with rf_1, then it holds with rf_2, too.

In the following lemma, we provide a sufficient condition for checking the dominance relation for request functions.

Lemma 3: A request function rf_1 dominates another request function rf_2 if the following conditions hold:

1) $\forall t : rf_1(t) \geq rf_2(t)$.
2) rf_1 and rf_2 contain the same sequence of actions.
3) (AS_{rf_i} is empty) or ($t_s \leq t'_s$ and $rf_1(t_s) \geq rf_2(t'_s)$ and rf'_1 dominates rf'_2), where $(s, t_s) = AS_{rf_1}[0]$, $(s, t'_s) = AS_{rf_2}[0]$, and rf'_1 and rf'_2 are obtained by $Align\_and\_Pop(rf_1, rf_2, s)$.

Proof: The proof is by induction on the number of actions contained in the request functions. Here, we give a sketch of the proof. For $|AS_{rf}| = 0$, the problem becomes the same as the problem for the DRT task model, which has been proven in [2]. If $|AS_{rf}| = 1$, then the conditions $t_s \leq t'_s$ and $rf_1(t_s) \geq rf_2(t'_s)$ guarantee that the alignment of any set of request functions with rf_1 specifies equal or more workload compared to when they are aligned with rf_2. In addition,

[2]Note that due to the inter-task synchronization, the $Synch()$ operation in (6) is applied to the set of request functions of all tasks (including the lower priority ones).

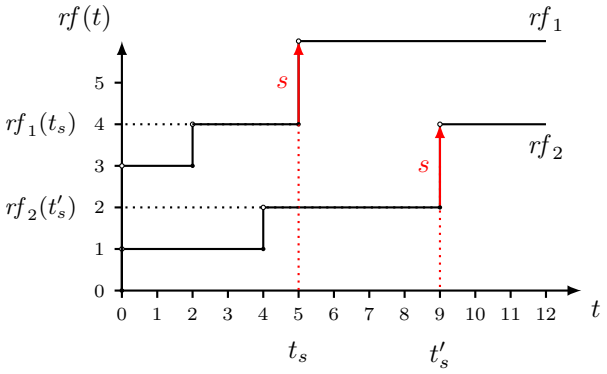


Fig. 8: Request function rf_1 dominates rf_2.

removing the actions leads to a problem with $|AS_{rf}| = 0$ which is already proven in the base case of the induction. Further, assume that the lemma holds for $|AS_{rf}| = n$. Similar justification can be made for $|AS_{rf}| = n + 1$ considering the alignment with respect to the first actions, and removing the actions after the alignment, which gives a problem instance with $|AS_{rf}| = n$. ■

Figure 8 shows an example of a request function (rf_1) which dominates another request function (rf_2).

B. Request Function Abstraction

Based on the condition provided in Theorem 2, to make sure that a task set is schedulable, the schedulability condition must be checked for all path combinations $\bar{\pi} \in \Pi(\tau)$. To avoid assessing all combinations, we incorporate the abstraction refinement approach [2], [23]. The main idea in this technique is that instead of exhaustively investigating the combination of the set of all *concrete* instances of the problem, we can start with an abstract level. More specifically, we can aggregate a set of request functions of a task such that they are over-approximated by an *abstract* request function. Then, if the schedulability condition holds for the abstract request function, we can infer that it holds for all ingredient request functions (namely those which are abstracted), without any need to check them individually. In the following, we formally define an abstract request function.

Definition 9 (Abstract Request Function): The abstract request function of a set of request functions $RF = \{rf_1, rf_2, \ldots, rf_n\}$ is a request function rf where

- $\forall t : rf(t) = max(rf_1(t), rf_2(t), ..., rf_n(t))$, and
- $AS_{rf} = \big\{(u, t_u) \mid \exists rf_i \in RF : (u, t_u) \in AS_{rf_i}$ *and* $t_u = \min\{t'_u \mid (u, t'_u) \in AS_{rf_j}$ for some $rf_j \in RF\}\big\}$

Additionally, any $rf_i \in RF$ is called an ingredient request function.

On the other hand, a request function is concrete if it is derived from a path in the respective graph. Based on the abstraction refinement approach, concrete and abstract request functions are considered in a binary tree structure, called *abstraction tree*, in which, all concrete request functions are placed in leaves. Further, each internal node is associated to the abstract request function obtained from abstraction of its children. Figure 9 provides a schematic view of the structure of such a tree.

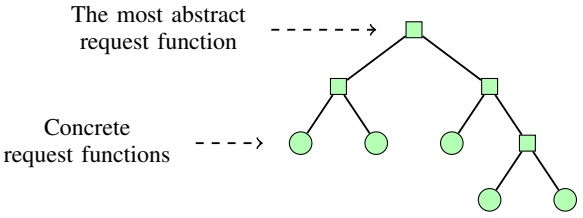


Fig. 9: A schematic view of the abstraction tree of a task, taken from [28].

function $Synch(R)$ ▷ R is a set of request functions
1: $R_a \leftarrow \{rf \in R \mid rf$ is abstract$\}$
2: $R_c \leftarrow \{rf \in R \mid rf$ is concrete$\}$
3: $Align(R_c)$
4: **while** True **do**
5: $S = \{ (rf, rf') \mid rf \in R_c, rf' \in R_a, AS_{rf}[0].s = AS_{rf'}[0].s, AS_{rf}[0].t < AS_{rf'}[0].t \}$
6: **if** S is empty **then**
7: break
8: **end if**
9: **for** each $(rf, rf') \in S$ **do**
10: $Align\_and\_Pop(rf, rf', AS_{rf}[0].s)$
11: **end for**
12: **end while**
13: **for** each $rf \in R_c$ **do**
14: **if** AS_{rf} is not empty **then**
15: $(s, t_s) \leftarrow AS_{rf}[0]$
16: **if** $\exists rf' \in R : (\exists t'_s : (s, t'_s) \in AS_{rf'})$ **then**
17: continue
18: **end if**
19: $new\_rf(t) = \begin{cases} rf(t), & \text{if } t < t_s, \\ rf(t_s), & \text{otherwise,} \end{cases}$
20: $rf \leftarrow new\_rf$
21: $AS_{rf} \leftarrow \{\}$
22: **end if**
23: **end for**

Fig. 10: Alignment of a mixed set of abstract and concrete request functions.

For schedulability analysis, we redefine the *Synch()* operation for a mixed set of abstract and concrete request functions as shown in Fig. 10. As seen, a *while* loop has been added (lines 4 to 12) to the procedure. In this loop, it is checked that whether an action in a concrete request function happens earlier compared to its appearance in an abstract one. If this is the case, we can align the former with respect to the latter. This is because of the definition of the action sequence of an abstract request function, which for any action, considers its earliest occurrence among the ingredient request functions. As a result, we know that if a concrete request function is to

function $Schedulable(\tau, v)$
1: $Q \leftarrow$ an empty queue
2: $r_i \leftarrow$ root of the abstraction tree obtained for task $T_i \in \tau$
3: $R \leftarrow (r_1, \ldots, r_n)$
4: $Q.add(R)$
5: **while** Q is not empty **do**
6: $R \leftarrow Q.pop()$
7: **if** $\forall t \leq d(v) : e(v) + \sum_{rf_i \in Synch(R),\ T_i \in \tau_{hp}} rf_i(t) > t$, **then**
8: **if** R is concrete **then**
9: **return** false
10: **else**
11: $Q.add(refine(R))$
12: **end if**
13: **end if**
14: **end while**
15: **return** true

Fig. 11: The procedure for schedulability analysis of a job v using the abstraction refinement technique.

be aligned with any of the ingredient ones, the earliest time at which the action can happen in a synchronous manner is considered. By this way, the aligned set of request functions provide an upper bound on the workload that can enter to the system up to each time t.

Figure 11 represents the procedure for schedulability analysis of a job using the abstraction refinement technique. In each step, the schedulability condition is checked on a combination of request functions (line 7). If the condition does not hold and all the request functions are concrete, then the job will be considered unschedulable (line 9). Meanwhile, if R contains at least one abstract request function, we need to refine it (because an abstract request function presents an over-approximation of the actual workload and we cannot conclude unschedulability of the job based on an abstract request function). In this case, the combination of request functions is refined by replacing one of the abstract ones with its children in the respective abstraction tree (line 11).

C. Action-Based Refinement

While the previous abstraction and refinement approach improves the efficiency of the analysis method, it does not scale very well when the number of actions is increased. (see Section VI for the experiment results illustrating this issue). In this section, we present an action-based abstraction approach to address this problem. This method is based on the intuition that by ignoring some actions, the task set can be approximated by another one with a simpler structure, and thus, a shorter analysis time. The details are presented in the following.

Consider an SDRT task set τ with the set of actions S. For any subset $S' \subseteq S$ we define task sets $\tau^-_{S'}$ and $\tau^+_{S'}$ as follows

$\tau^-_{S'} \equiv$ The task set obtained from τ by removing all the edges e for which $a(e) \in S'$.

$\tau^+_{S'} \equiv$ The task set obtained from τ by removing all the actions $s \in S'$ (without removing the edges).

We refer to S' as the set of ignored actions. Our approach is based on the relationship between τ, $\tau^-_{S'}$, and $\tau^+_{S'}$ from a the schedulability point of view, as specified in the following lemma.

Lemma 4: Consider an SDRT task set τ and an arbitrary set of ignored actions S'. Then, the following holds:

$$\tau^-_{S'} \text{ is unschedulable } \Rightarrow \tau \text{ is unschedulable} \quad (7)$$

$$\tau^+_{S'} \text{ is schedulable } \Rightarrow \tau \text{ is schedulable} \quad (8)$$

Proof: The proof of (7) is based on the fact that the set of all job sequences which can be generated by $\tau^-_{S'}$ is a subset of that of τ. This is because that removing a number of edges from an SDRT task graph removes some execution paths. On the other hand, it does not add any new execution trace. For the proof, assume that $\tau^-_{S'}$ is unschedulable. This means than there exists a job sequence under which at least one job misses its deadline. According to the mentioned fact, such a job sequence can be generated by τ as well, leading to a deadline miss. As a result, τ turns out to be unschedulable. A similar argument holds for (8) considering that all job sequences generated by τ can be generated by $\tau^+_{S'}$. ■

The action-based abstraction approach is constructed on the basis of Lemma 4. In this approach, instead of directly analyzing the task set τ, we proceed on the corresponding under- and over-approximations, namely $\tau^-_{S'}$ and $\tau^+_{S'}$. We begin by considering the largest set of ignored actions, namely $S' = S$, which provides the highest level of abstraction. In each step, schedulability of $\tau^+_{S'}$ and unschedulability of $\tau^-_{S'}$ are inspected. If $\tau^+_{S'}$ is schedulable, it is implied that τ is schedulable and the test is terminated. Besides, if $\tau^-_{S'}$ is unschedulable we can conclude that τ is unschedulable. Otherwise, we proceed one refinement step by removing one item from the set of ignored actions. In the worst scenario, we will need to remove all items from the set of ignored actions, achieving $S' = \emptyset$. At this step, $\tau^-_{S'} = \tau^+_{S'} = \tau$, which means that we have to analyze the actual task set for schedulability test. Figure 12 shows a schematic view on the refinement procedure. As seen, at each step, $\tau^-_{S'}$ and $\tau^+_{S'}$ are refined to provide more accurate approximations of τ.

We illustrate our action-based abstraction refinement approach using a simple example. Figure 13 shows a set of two SDRT tasks with two synchronizations. The abstraction and refinement steps related to this sample task set are shown in Fig. 14. In the first step, the set of ignored actions is determined as $S' = \{s_1, s_2\}$. The first column of the table depicts the respective task sets $\tau^-_{S'}$ and $\tau^+_{S'}$ for this step. In the next step, the task sets are refined by adding the synchronization action s_1. The results are seen in the second column of the table. Finally, by reviving action s_2, the task sets are completely refined, obtaining the original task set τ.

As a generalization of this simple example, the refinement is performed via a loop. At each iteration, the analysis method provided in the previous sections (including the abstraction

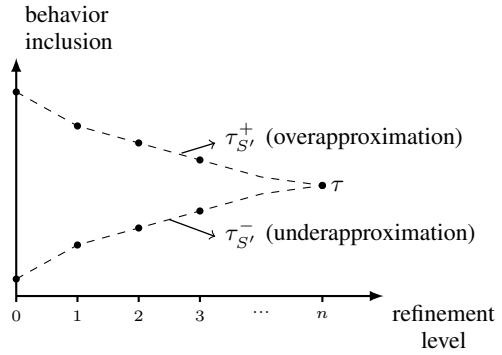


Fig. 12: A schematic view of the action-based abstraction refinement.

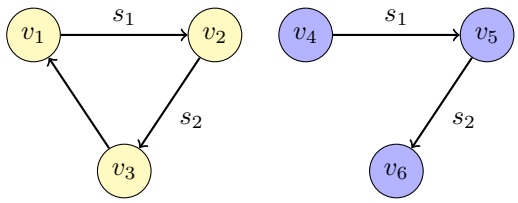


Fig. 13: A set of two simple SDRT tasks (for clarity, the vertex labels and inter-separation times are not shown).

refinement approach specified in Sec. V-B) is called on the task sets $\tau^-_{S'}$ and $\tau^+_{S'}$. If the analysis reveals that $\tau^-_{S'}$ is unschedulable (or $\tau^+_{S'}$ is schedulable), then it is concluded that the original task set (τ) is also unschedulable (or schedulable) and the loop is terminated. Otherwise, task sets $\tau^+_{S'}$ and $\tau^-_{S'}$ are refined one step by removing one synchronization action from S' (as seen in different steps in Fig. 14).

VI. Evaluation

We have implemented the proposed method using the python library previously developed for schedulability analysis of DRT task sets [28]. For random task set generation, similar to [2], we considered three types of tasks, namely small, medium, and large tasks, with the parameter ranges given in Table I. For each task, one of these types is randomly selected. Then, the task parameters are chosen from the corresponding intervals with a uniform probability. In addition, synchronization actions are added to the randomly selected edges. Regarding the task synchronization, we study two cases. In the first one, n synchronization actions are added to the task set, where n is the number of tasks in the task set. In the second case, $3n$ actions are added, which represents a relatively high degree of synchronization between the tasks. In addition, we obtained the results for the task sets containing no action. In this case, the methods behave similar to which proposed for the DRT tasks [2]. In order to generate a task set with a desired utilization, random tasks are generated and added to the task set until the total utilization of the task set becomes larger than the specified utilization. As a consequence, task sets with higher utilizations are usually associated with higher number of tasks. In our experiments, for each data point, 100 random experiments have been run.

TABLE I: Task set parameters

Task Type	Small	Medium	Large
Vertices	$[3, 5]$	$[5, 9]$	$[7, 13]$
Branching degree	$[1, 3]$	$[1, 4]$	$[1, 5]$
p	$[50, 100]$	$[100, 200]$	$[200, 400]$
e	$[1, 2]$	$[1, 4]$	$[1, 8]$
d	$[25, 100]$	$[50, 200]$	$[100, 400]$

We first explore that how much the dominance relation helps in state-space reduction. To this aim, we report the number of path combinations which should be considered in the schedulability analysis. The results are depicted in Fig. 15. As seen, the reduction obtained by the dominance relation is considerable compared to the total combinations. It is also observed that in task sets with higher number of actions less request functions are dominated. This is expected because, according to Definition 8, when the number of actions increases (or equivalently, $|AS_{rf}|$ is larger, on the average), the dominance conditions are less likely to hold.

We have assessed the scalability of the proposed methods through varying the total number of actions in a task set. We have changed the number of synchronization actions from two to 40 with a step of two. For each experiment, we generated a task set with the desired utilization, and then, run the analysis method to test schedulability of a single job with the lowest priority which has no synchronization with the higher priority tasks (the higher priority tasks have synchronizations with each other). We obtained the average run-time for two cases: (i) the analysis *without* the action-based abstraction refinement; and (ii) the analysis *with* the action-based abstraction refinement. Figure 16 shows the results for two values of the task set utilization, namely 0.5 and 0.6. As seen, increasing the number of actions in the task sets causes that the analysis approach (without the action-based refinement) become very lengthy. In contrast, the action-based abstraction refinement approach scales very well when the number of actions is increased. This is because that this method adds the actions gradually and often it reaches a decision on the schedulability (or unschedulability) of the task set earlier.

Further, we investigate the acceptance ratio, as well as the number of actual combinations which are tested by the action-based abstraction refinement method, for task sets with different number of synchronization actions. Acceptance ratio is computed by dividing the number of schedulable task sets by the total number of task sets. We used the Audsley's algorithm [29] for SP schedulability of a task set. The results are presented in Table II. As seen, task sets with higher number of actions exhibit higher acceptance ratio. This is because that the synchronization requirement restricts the tasks, leading to equal or less workload released until any time instant and a lower chance of deadline miss. In addition, it is seen that with

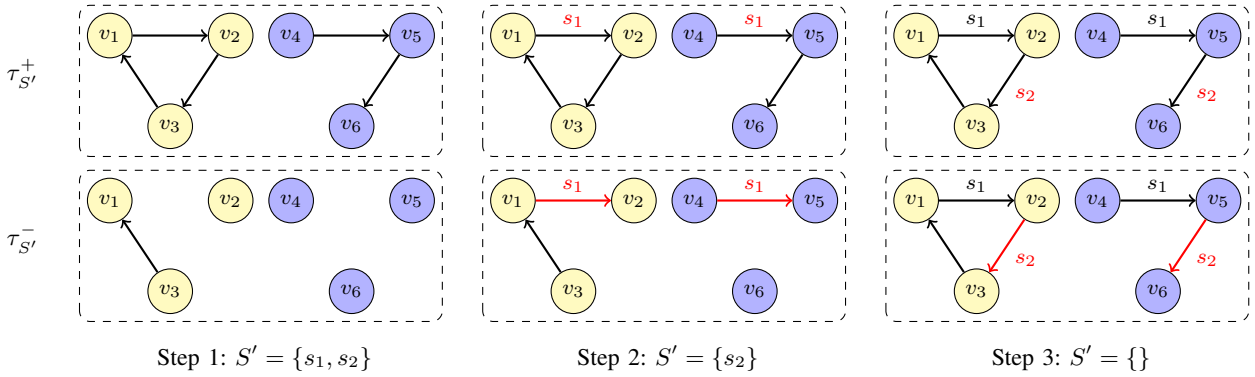


Fig. 14: Action-based abstraction refinement for the task set τ specified in Fig. 13.

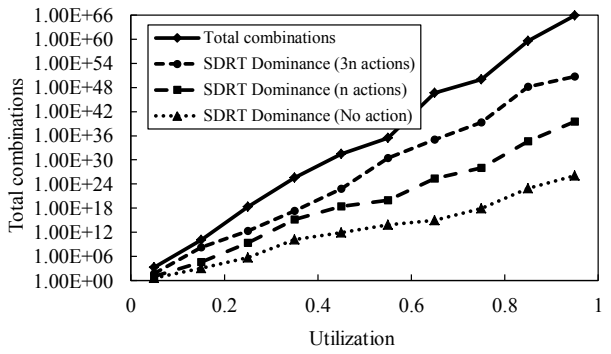


Fig. 15: Number of path combinations that must be considered in schedulability analysis.

TABLE II: Schedulability analysis results

	Acceptance Ratio			Tested Combinations		
Util.	No act.	n act.	$3n$ act.	No act.	n act.	$3n$ act.
0.35	1	1	1	20	22	22
0.4	1	1	1	29	29	29
0.45	0.99	0.99	0.99	47	51	48
0.5	0.91	0.92	0.96	80	150	20860
0.55	0.44	0.52	0.58	153	958	387167
0.6	0.15	0.19	0.26	199	1603	128074
0.65	0.01	0.01	0.02	172	29872	19167

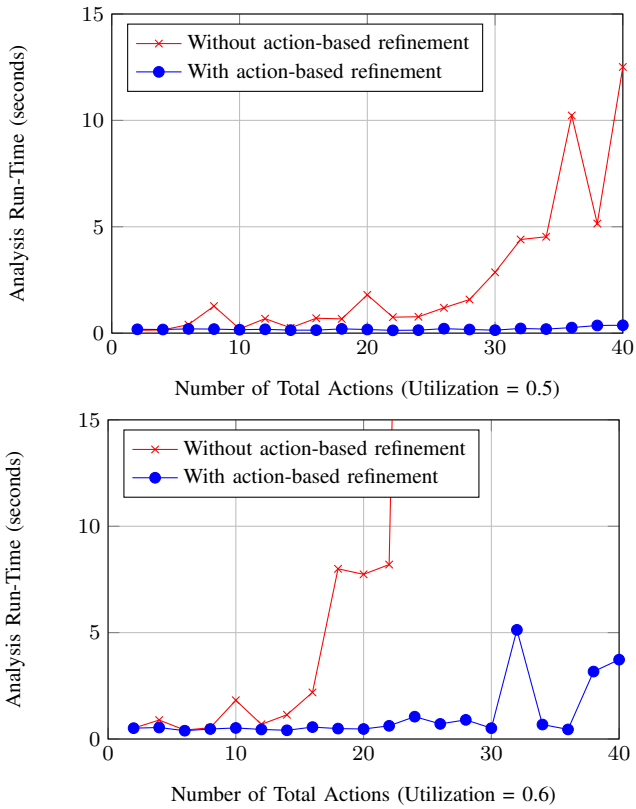


Fig. 16: Average run-time of the proposed methods.

increased number of actions, more combinations are checked during the analysis. There are two reasons for this result. First, as shown in Fig. 15, total combinations increase with higher number of actions. Second, with more actions, the abstract request functions provide less accurate estimate of the ingredient ones, reducing the effectiveness of the approach.

VII. Conclusion

In this paper, we proposed an extension of the DRT task model through which the inter-task synchronization (rendezvous) can be specified. We proposed a method for SP schedulability analysis of this model for uniprocessor systems. We also introduced two abstraction refinement techniques for improving the efficiency of the analysis. The extension of the proposed approach to the case of dynamic priority scheduling is regarded as a future work. In addition, the capability of the proposed model for specifying task sets described by other existing models (such as the fork-join model [15]) can be investigated.

Acknowledgment

The authors would like to thank Pontus Ekberg for his valuable comments.

REFERENCES

[1] M. Stigge, P. Ekberg, N. Guan, and W. Yi, *The digraph real-time task model*. In: Proceedings of Real-Time and Embedded Technology and Applications Symposium, pp. 71–80, 2011.

[2] M. Stigge and W. Yi, *Combinatorial abstraction refinement for feasibility analysis*. In: Proceedings of Real-Time Systems Symposium (RTSS), pp. 340–349, 2013.

[3] N. Guan, C. Gu, M. Stigge, Q. Deng, and W. Yi, *Approximate response time analysis of real-time task graphs*. In: Proceedings of Real-Time Systems Symposium (RTSS), pp.304–313, 2014.

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, *Generalized multiframe tasks*. Real-Time Systems, 17(1): pp. 5–22. (1999)

[5] A. K. Mok and D. Chen, *A multiframe model for real-time tasks*. IEEE Transactions on Software Engineering, 23(10): pp. 635–445. (1997)

[6] N. T. Moyo, E. Nicollet, F. Lafaye, and C. Moy, *On schedulability analysis of non-cyclic generalized multiframe tasks*. In: Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), pp. 271–278, 2010.

[7] S. K. Baruah, *Feasibility analysis of recurring branching tasks*. Euromicro Workshop on Real-Time Systems, pp. 138–145, 1998.

[8] S. K. Baruah, *Dynamic- and static-priority scheduling of recurring real-time tasks*. Real-Time Systems, 24(1): pp. 93–128. (2003)

[9] C. A. R Hoare, *Communicating sequential processes*. Communications of the ACM, 21(8): pp. 93–128. (1978)

[10] C. A. Petri, *Kommunikation mit Automaten*. PhD thesis, Institut fur instrumentelle Mathematik, Bonn. (1962)

[11] J. Barnes, *Programming in Ada 2005*. Addison-Wesley, ISBN 0-321-34078-7. (2006)

[12] A. W. Roscoe and C. A. R. Hoare, *The laws of occam programming*. Programming Research Group, Oxford University (1986)

[13] S. A. Edwards and O. Tardieu, *SHIM: A deterministic model for heterogeneous embedded systems*. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14(8): pp. 854–867. (2005)

[14] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. R. Sachs, and Y. Xiong, *Taming heterogeneity - the Ptolemy approach*. Proceedings of the IEEE 91(1): pp. 127–144. (2003)

[15] P. Axer, S. Quinton, M. Neukirchner, R. Ernst, B. Dobel, and H. Hartig, *Response-time analysis of parallel fork-join workloads with real-time constraints*. In: Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), pp. 215–224, 2013.

[16] K. G. Larsen, P. Pettersson, and W. Yi, *UPPAAL in a nutshell*. Software Tools for Technology Transfer, 1(1/2): pp. 134–152. (1997)

[17] Sanjoy K. Baruah, *The non-cyclic recurring real-time task model*. In: Proceedings of Real-Time Systems Symposium (RTSS), pp. 173–182, 2010.

[18] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, *TIMES - A tool for modelling and implementation of embedded systems*. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 460–464, 2002.

[19] A. K. Mok, *The design of real-time programming systems based on process models*. In: Proceedings of Real-Time Systems Symposium (RTSS), pp. 5–17, 1984.

[20] S. Ramamurthy, *The design of real-time programming systems based on process models*. Ph.D. thesis at University of North Carolina at Chapel Hill. (1997)

[21] E. W. Giering and T. P. Baker, *Toward the deterministic scheduling of Ada tasks*. In: Proceedings of Real-Time Systems Symposium (RTSS), pp. 31–40, 1989.

[22] N. Guan, P. Ekberg, M. Stigge, W. Yi, *Resource sharing protocols for real-time task graph systems*. In: Proceedings of Euromicro Conference on Real-Time Systems (ECRTS), pp. 272–281, 2011.

[23] N. Guan, Y. Tang, J. Abdullah, M. Stigge, and W. Yi, *Scalable timing analysis by refinement*. In: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 3–18, 2015.

[24] M. Stigge, P. Ekberg, and W. Yi, *The fork-join real-time task model*. In: SIGBED Rev. 10, 2 , pp. 20–20. (2013)

[25] C. L. Liu and J. W. Layland, *Scheduling algorithms for multiprogramming in a hard-real-time environment*. Journal of the ACM 20(1): pp. 46–61. (1973)

[26] A. K. Mok, *Fundamental design problems of distributed systems for the hard-real-time environment*. Technical report, Cambridge, MA, USA, 1983.

[27] J. W. McCormick, F. Singhoff, and J. Hugues, *Building parallel, embedded, and real-time applications with Ada*. Cambridge University Press, 2011.

[28] M. Stigge, *Real-time workload models: Expressiveness vs. analysis efficiency*. PhD Thesis, Uppsala University. (2014)

[29] N. C. Audsley, *Optimal priority assignment and feasibility of static priority tasks with arbitrary start times*. University of York. (1991)

[30] M. Mohaqeqi, J. Abdullah, N. Guan, and W. Yang, *Schedulability Analysis of Synchronous Digraph Real-Time Tasks*. Digitala vetenskapliga arkivet (DiVA), Uppsala University. (2016)

APPENDIX A
SCHEDULABILITY ANALYSIS FOR THE GENERAL CASE

In this section, we present schedulability analysis for a job in a general case where the release of that job may be synchronized with a release of another job (from a different task). For this purpose, first we need to argue that what the worst-case arrival pattern of a set of SDRT tasks on a job is. By the worst-case arrival pattern (usually referred to as the *critical instant*), we mean the arrival pattern, i.e. the set of job sequences, which leads to the maximum interference, or equivalently the maximum response time, of a specific job. This argument is important since in the schedulability analysis of SDRT tasks we need to know the worst-case arrival pattern. In particular, the correctness of Theorem 2 depends on such analysis.

Assume that we want to test the schedulability of a job v which belongs to a task τ_i. The interfering workload on v from the higher priority tasks consists of two parts: (i) the workload corresponding to the jobs released simultaneously with or after v, and (ii) the workload related to those jobs which arrived before v. We show that when the release of v is not synchronized with any other job, it suffices to consider only the first part. On the other hand, if v is subject to a synchronous release, then we must take into account both parts.

In the following, first some preliminary definitions and lemmas are presented. Then, we investigate each of the mentioned cases in a separate subsection.

A. Preliminaries

First of all, we recall that a job is denoted by a tuple (R_i, e_i, v_i) where R_i, e_i, and v_i represent the job arrival time, its execution time, and the job type, respectively. Also, $\sigma_i = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ denotes a job sequence generated by a task if the release times, execution times and the job types comply with the respective task graph.

Let $\tau = \{\tau_1, \ldots, \tau_n\}$ be a set of tasks. Also, define

$$\begin{aligned} \sigma_{\tau_i} &\equiv \text{The set of job sequences generated by task } \tau_i, \\ \sigma_{\tau} &\equiv \{(\sigma_1, \ldots, \sigma_n) \mid \forall i : \sigma_i \in \sigma_{\tau_i}, \text{ and } \{\sigma_1, \ldots, \sigma_n\} \\ &\qquad \text{is a set of synchronous job sequences (SJS)}\}. \end{aligned}$$

In words, σ_τ denotes the set of all synchronous job sequences, which is obtained from $\sigma_{\tau_1} \times \ldots \times \sigma_{\tau_n}$ after removing all non-synchronous set of job sequences (for the definition of synchronous job sequences, see Definition 3).

Further, for a particular job sequence $\sigma_i = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$, we define $I_{\sigma_i}(t_1, t_2)$ as

the accumulated workload generated by the respective task in an interval $[t_1, t_2]$, formally defined as

$$I_{\sigma_i}(t_1, t_2) = \sum_{t_1 \le R_j < t_2} e_j \tag{9}$$

It is seen that $I_{\sigma_i}(t_1, t_2)$ preserves the following properties:

$$\begin{aligned} I(t_1, t_1) &= 0 \\ I(t_1, t_2) &= I(t_1, t) + I(t, t_2),\ \forall t \in [t_1, t_2] \end{aligned} \tag{10}$$

Here, we introduce a modified notion of *job sequence shifting* which is slightly different from which defined in Definition 4.

Definition 10 (Job Sequence Shifting): Consider a job sequence $\sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots]$ generated by task T. In addition, let t and δ be two arbitrary positive integers. Then, the *shifted* job sequence with respect to t and δ is defined as the job sequence $\sigma'(t, \delta) = [(R'_0, e_0, v_0), (R'_1, e_1, v_1), \ldots]$ where

$$R'_j = \begin{cases} R_j, & R_j < t, \\ R_j + \delta, & R_j \ge t, \end{cases}$$

As an example, imagine a job sequence with the workload shown in Fig. 19. The arrival pattern shown in Fig. 20 can be considered as a workload obtained by shifting the workload of Fig. 19 by parameters $t = -3$ and $\delta = 3$.

It is observed that when a set of SJSs are shifted with the same shifting parameters t and δ, the obtained job sequences are a set of SJSs as well. This is formally specified in the following lemma.

Lemma 5: Consider a task set τ and a set of SJS $(\sigma_1, \ldots, \sigma_n) \in \sigma_\tau$ generated by τ. Then, for any $t, \delta \ge 0$, the shifted set of job sequences

$$\sigma' = (\sigma'_1(t, \delta), \ldots, \sigma'_n(t, \delta))$$

is also a set of SJS generated by τ, that is $\sigma' \in \sigma_\tau$. In addition, for any t_1 and t_2 with $t \le t_1 \le t_2$, the workload generated by the tasks with the shifted job sequences in the time interval $[t_1 + \delta, t_2 + \delta]$ is equal to that of the original job sequences in $[t_1, t_2]$. More formally, for all τ_i, and for any t_1, t_2 with $t \le t_1 \le t_2$, we can write

$$I_{\sigma'_i(t,\delta)}(t_1 + \delta, t_2 + \delta) = I_{\sigma_i}(t_1, t_2). \tag{11}$$

Proof: According to the definition of synchronous job sequences (Definition 3) and the definition of the action sequence corresponding to a job sequence (Definition 2), it is revealed that the shifted set of job sequences is synchronous. In addition, according to the definition of $\sigma'_i(t, \delta)$, it is seen that the jobs released by $\sigma'_i(t, \delta)$ in $[t_1 + \delta, t_2 + \delta)$ are exactly the same jobs (with the same execution times, job type, inter-release times, and the same order) which are released by σ_i in $[t_1, t_2)$, given that $t \le t_1 \le t_2$. In other words, for any job (R_j, e_j, v_j) in σ_i with $t_1 \le R_j < t_2$, there exists exactly one job in $\sigma'_i(t, \delta)$, i.e. (R'_j, e_j, v_j) where $R'_j = R_j + \delta$. Also, we have $t_1 \le R_j < t_2$ if and only if $t_1 + \delta \le R'_j < t_2 + \delta$. As a consequence, and based on the definition of $I_{\sigma_i}(.,.)$ in (9), the proof is completed. ■

Further, we extend the lemma based on the observation that in shifting a set of SJS, a job release which is not associated with any synchronization action does not influence the result of the lemma. This is specified in the following lemma.

Lemma 6: Consider a task set τ and a set of SJS $(\sigma_1, \ldots, \sigma_n) \in \sigma_\tau$ generated by τ. Assume that, for a job $(R_j, e_j, v_j) \in \sigma_i$ in some σ_i, it holds that

$$\forall (s, t_s) \in AS_{\sigma_i} : t_s \ne R_j,$$

that is, no synchronization is associated with (R_j, e_j, v_j). Then, for any t with $R_{j-1} < t < R_j$, the shifted set of job sequences

$$\sigma' = (\sigma'_1(t, \delta), \ldots, \sigma'_{i-1}(t, \delta), \sigma'_i(R_{j+1}, \delta), \sigma'_{i+1}(t, \delta), \ldots, \sigma'_n(t, \delta))$$

is also a set of SJS generated by τ, that is $\sigma' \in \sigma_\tau$, for any $\delta > 0$. In other words, when a job release (i.e., (R_j, e_j, v_j)) is not synchronized with the release of any other job, shifting the corresponding job sequence can be *delayed* until the release of the next job of the same task, without affecting the synchrony of the whole set of job sequences. Additionally, for any t_1 and t_2 satisfying $t \le t_1 \le t_2$, we can write

$$I_{\sigma'_j(t,\delta)}(t_1 + \delta, t_2 + \delta) = I_{\sigma_j}(t_1, t_2). \tag{12}$$

for all $\tau_j \in \tau \backslash \tau_i$ [3].

Proof: By using Lemma 1 it is seen that the mentioned set of job sequences, i.e., σ', can be generated by the task set. Further, regarding the definition of a SJS which is based on the pair-wise equality of the action sequences of the tasks, σ' is also synchronous. (this is based on the fact that if we shift two action sequences by the same amount, then if the original ones are synchronous, the shifted ones are also synchronous.)

Also, for $\tau_j \ne \tau_i$, the situation is exactly the same as in Lemma 5. Therefore, (12) is immediately inferred from (11). ■

Lemma 7: Consider a job v. Let v_{-1} denote the latest job of the same task of v which has been released before v. Then, given that v_{-1} is schedulable, the workload (of the higher priority tasks) existing in the system at the release time of v have arrived after the completion of v_{-1}.

Proof: This is because that otherwise, v_{-1} is not finished yet, leading to its deadline miss (as we have assumed the constrained deadlines). ■

We also define the maximum busy interval of a job with respect to a given set of job sequences as follows.

Definition 11 (Maximum Busy Interval): Consider a set of job sequences $\sigma = (\sigma_1, \ldots, \sigma_n)$ and a job (R_j, e_j, v_j) in some σ_i. The maximum busy interval of this job under σ is the maximal interval $I = [I_s, I_e]$ that satisfies the following constraints:

$$R_{j-1} < I_s \le R_j < I_e, \tag{13a}$$

$$\forall t \in (I_s, I_e) : \sum_{\tau_j \in \tau_{hp}} I_{\sigma_j}(I_s, t) > t - I_s, \tag{13b}$$

[3] $\tau \backslash \tau_i$ is defined as $\tau \backslash \tau_i \equiv \{\tau_j \in \tau \mid \tau_j \ne \tau_i\}$.

where τ_{hp} denotes the set of task with priority higher than τ_i. Figure 19 illustrates a visual representation of such an interval.

Now, we are ready to proceed with the discussion on the critical instant for an SDRT job.

B. Non-Synchronous Job

In this section, we consider the schedulability analysis of a job assuming that its release is not synchronized with the release of any other job.

Definition 12 (Non-Synchronous Job): Take an SDRT task τ_i. A job type $v \in V(\tau_i)$ is said to be non-synchronous if and only if

$$\forall (u,v) \in E(\tau_i) : a(u,v) = \perp$$

We now formally define the response time of a job.

Definition 13 (Response Time): Consider a job $J = (R_j, c, v_j)$ with an execution time of $c > 0$. The response time of J with respect to a set of SJS $\sigma \in \sigma_\tau$ is defined as:

$$RT_\sigma(J) \equiv \min_{t>0} t : c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(I_s, I_s + t) \leq t, \qquad (14)$$

where I_s is the beginning of the maximum busy interval of the job, and $I_{\sigma_i}(I_s, t)$ is the accumulated workload generated by σ_i as defined in (9).

Next, we specify a useful property related to the response time of a non-synchronous job.

Lemma 8: Take a set of SJS $\sigma = (\sigma_1, \ldots, \sigma_n)$ and assume that $J = (R_j, e_j, v_j)$ is a job in some σ_i where v_j is a non-synchronous job (see Definition 12). In addition, assume that $I = [I_s, I_e]$ is the maximum busy interval of J. Then, it holds that

$$RT_\sigma(J) \leq RT_{\sigma'(I_s,\delta)}(J), \qquad (15)$$

where $\delta = R_j - I_s$, and $\sigma'(I_s, \delta)$ denotes the shifted set of job sequences as described in Lemma 6 (i.e. shifting σ_i is delayed).

Proof: First, let r_1 be the response time of J under σ, that is,

$$r_1 = \min_{t>0} t : c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(I_s, I_s + t) \leq t.$$

This means that,

$$\forall t \in [0, r_1) : c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(I_s, I_s + t) > t. \qquad (16)$$

In order to investigate the correctness of (15), we proceed in two steps. First, we show that the response time cannot be t for $0 \leq t \leq \delta$. To show this, we write

$$\begin{aligned} c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + t) &> \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + t) \\ &= \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(I_s, I_s + t) \\ \text{(because of (13b))} &> t, \end{aligned} \qquad (17)$$

where $I_{\sigma'_i(I_s,\delta)}(.,.)$ has been abbreviated as $I_{\sigma'_i}(.,.)$ for notation brevity. This is obtained from the assumption $c > 0$, and from (12) and (13b). This reveals that the response time of v_j is strictly larger than δ. Now, we consider the situation for t where $\delta < t < r_1$. Regarding this assumption, one can write

$$\begin{aligned} c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + t) &= c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + \delta) + \\ \text{(from (10))} & \quad \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j + \delta, R_j + t) \\ &= c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(I_s, R_j) + \\ & \quad \sum_{\tau_i \in \tau_{hp}} I_{\sigma_i}(R_j, R_j + t - \delta) \\ \text{(from (16))} &> (R_j - I_s) + (t - \delta) \\ &= \delta + (t - \delta) \\ &= t \end{aligned} \qquad (18)$$

Therefore, we have

$$c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + t) > t \qquad (19)$$

for $\delta < t < r_1$. As a result of (17) and (19), we have

$$\forall t \in (0, r_1) : c + \sum_{\tau_i \in \tau_{hp}} I_{\sigma'_i}(R_j, R_j + t) > t \qquad (20)$$

which means that the response time of J under the shifted set of job sequences cannot be less than r_1, which completes the proof. ■

Theorem 3: Consider a non-synchronous job v released at t_0 and a set of n higher priority tasks τ_{hp}. Suppose $\sigma = (\sigma_1, \ldots, \sigma_n)$ is a critical instant (i.e. the job sequence with the maximum interfering workload) for v generated by τ_{hp}. Then, there exists a job sequence $\sigma' = (\sigma'_1, \ldots, \sigma'_n)$ such that

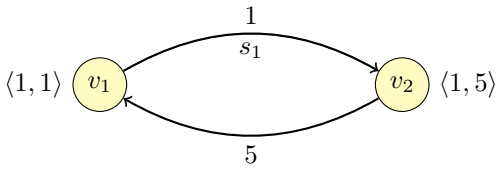
- All job sequences σ'_i, for $1 \leq i \leq n$, start with or after v, i.e.,
$$\forall \sigma'_i = [(R_0, e_0, v_0), \ldots] : t_0 \leq R_0$$
- σ' entails a larger or equal response time to v than σ.

Proof: This is implied from Lemma 6 with the shifting parameters $t = R_j$ and $\delta = t_0 - I_s$. ■

An important result of this theorem is that for schedulability analysis of a non-synchronous job J, it is sufficient to only consider the set of SJSs for which the maximum busy interval of v starts with the release time of J. In the next subsection, we show that this result does not necessarily hold for synchronous jobs.

C. Synchronous Job

In this section, the schedulability analysis of a synchronous job, namely a job which must be released synchronous with some other job, is considered. We first provide an example in which the response time of a job v with release time of t_0 regarding a set of interfering job sequences which produce some workload before t_0 is larger than any other arrival pattern in which all jobs release at or after t_0. This example shows that Theorem 3 does not hold for synchronous jobs. Then, we show that how this case can be considered in the schedulability analysis.



(a) Task τ_1

(b) Task τ_2

Fig. 17: Two sample SDRT tasks.

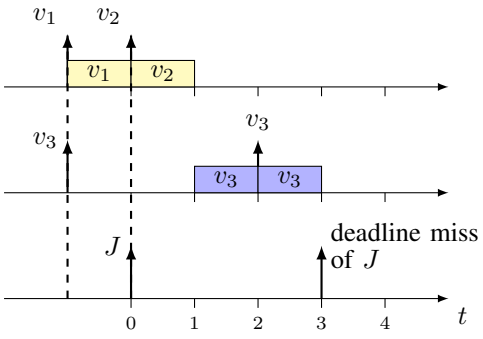


Fig. 18: Sample schedule.

Example 3: Consider two SDRT tasks τ_1 and τ_2 specified in Fig. 17. In addition, let J be a job with a priority lower than τ_1 and τ_2. Assume the relative deadline of J is 3. Also, suppose that the ingoing edge to the J's vertex in the respective task graph is labeled with s_1. This means that the release of J must be synchronized with the release of v_2 from τ_1. Figure 18 shows a possible schedule for this task set. As seen, the represented arrival pattern leads to a deadline miss of J. On the other hand, it is seen that no synchronous arrival pattern (namely a job sequence where all the jobs are released at $t = 0$ or later) can produce such a workload in which J misses its deadline.

This counter example implies that for schedulability analysis of a synchronous job J, we need to consider not only those set of SJS which begin with the release time of J, but also those that start before the arrival time of J. In this case, the problem is to find an upper bound for the length of the maximum busy intervals that are required to be checked for schedulability anlaysis. Without such an upper bound, there can be infinitely many set of job sequences which should be considered for schedulability test of a single job.

To address the mentioned problem, we first define the notion of *duration* for a job sequence.

Definition 14 (Job Sequence Duration): Assume $\sigma_i = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots, (R_{n_i}, e_{n_i}, v_{n_i})]$ be a finite job sequence. The duration of σ_i is defined as

$$D(\sigma_i) = R_{n_i} - R_0$$

In addition, the duration of a set of SJS $\sigma = (\sigma_1, \ldots, \sigma_n)$ is defined as

$$D(\sigma) = \max_{1 \le i \le n} D(\sigma_i)$$

We notice that there are two scenarios for the release of a synchronous job $J = (R, e, v)$: (i) J is the first job released by the respective task or its previous job is of type u for which $a(u, v) = \perp$. In this case, the release of J is not synchrnoized with the release of any other job. This is referred to as the *non-synchronous release scenario*; (ii) J is released synchrnoized with another job. This case is referred to as the *synchronous release scenario* of J.

Lemma 9: Consider a job $J = (R, e, v)$ and an arbitrary maximum busy interval $[I_s, I_e]$ for it with $I_s < R$. Then, none of the jobs (of the other tasks) released in interval $[I_s, R)$ are synchronized with J.

Proof: Based on Lemma 7, for any job $J = (R, e, v)$ and any maximum busy interval $[I_s, I_e]$ associated with J, no job of the same task of J can be released in interval $[I_s, R)$. As a result, J itself is not released during $[I_s, R)$, which means that any job synchronous with J cannot appear in this interval. ■

For schedulability analisys of a job $J = (R, e, v)$ we consider two cases. First, we assume that there exists a maximum busy interval $[I_s, I_e]$ for J associated with a set of SJS σ such that $R - I_s \ge d$, where d denotes the relative deadline of J. Based on Lemma 9, the jobs in σ which are released during $[I_s, R)$ constitute a non-synchronous release scenario for J. As a result, there exists a non-synchronous release scenario which leads to a deadline miss of J. Therefore, the method used for the situation of a non-synchronous job elaborated in the previous section suffices to detect this deadline miss of the job.

Now, we focus on the other case, namely the scenario in which

$$R - I_s < d \tag{21}$$

For response time analysis of J, we know that it is sufficient to check the respective condition (i.e. Eq. (14)) only up to the deadline of J (that is, $R + d$) [4]. As a result, in the evaluation of the workload functions in the mentioned relation, i.e. $I_{\sigma_i}(I_s, I_s + t)$, we need to only check those values of t for which

$$t + I_s \le R + d, \tag{22}$$

or equivalently

$$t \le R + d - I_s. \tag{23}$$

This means that we must consider job sequences with a maximum duration of $t \le R + d - I_s$. Merging this relation with (21) reveals that, an upper bound for the length of the interval for which the workload functions should be evaluated can be computed as $t \le R + d - I_s < R + d + d - R = 2d$. This gives an upper bound for the duration of the set of SJS which should be checked for schedulability of an SDRT job.

[4]This is because that if there is no time instant t between the release time and deadline of J which satisfies the inequality in (14), then the response time of J will be obviously larger than its deadline, which means a deadline miss.

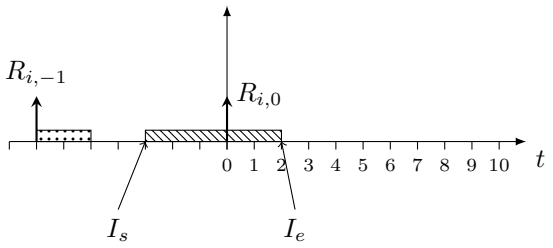


Fig. 19: Original sequence of the job releases with remaining workload at the release time of J (namely at $t = 0$).

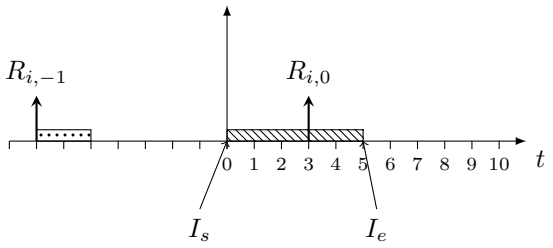


Fig. 20: Postponing a job release for δ time units.