This paper is published in Real-Time Systems. The final publication is available at http://link.springer.com.

Combinatorial Abstraction Refinement for Feasibility Analysis of Static Priorities

Martin Stigge · **Wang Yi**

Abstract Combinatorial explosion is a challenge for many analysis problems in the theory of hard real-time systems. One of these problems is static priority schedulability of workload models which are more expressive than the traditional periodic task model. Different classes of directed graphs have been proposed in recent years to model structures like frames, branching and loops. In contrast to dynamic priority schedulers with pseudo-polynomial time analysis methods, static priority schedulability has been shown to be intractable since it is strongly coNP-hard already for the relatively simple class of cyclic digraphs. The core of this problem is the necessity to combine different behaviors of the participating tasks.

We introduce a novel iterative approach to efficiently cope with this combinatorial explosion, called combinatorial abstraction refinement. In combination with other techniques it significantly reduces exponential growth of run-time for most inputs. We apply the method to static task priorities and demonstrate that a prototype implementation outperforms the state-of-the art pseudo-polynomial analysis for dynamic priority feasibility. It further shows better scaling behavior for randomly generated problem instances. We extend the approach to non-preemptive schedulers as well as static job-type priorities where jobs of different types in the same task may be assigned different static priorities. Finally, we provide a general, abstract formulation of the algorithm, since we believe that this method can be applicable to a variety of combinatorial problems in the theory of real-time systems with certain abstraction structures.

Keywords Real-time · Schedulability analysis · Static priorities · Combinatorial explosion · Abstraction refinement

M. Stigge · W. Yi
Uppsala University, Department of Information Technology, Box 337, SE-751 05 Uppsala, Sweden
E-mail: martin.stigge@it.uu.se

W. Yi
E-mail: yi@it.uu.se

1 Introduction

One of the core objectives in the theory of hard real-time systems is to analyse whether a given system workload will meet all its timing requirements at run-time for all conceivable situations. The classical *periodic task model* by Liu and Layland (Liu and Layland 1973) describes system workload as a collection of independent tasks. Each task is activated periodically and the run-time of each activation is bounded by a value derived from a worst-case execution time analysis.

The model's assumption that each task is behaving periodically is often too simple. More expressive models have been proposed in recent years (Mok and Chen 1997; Baruah et al. 1999; Baruah 1998, 2010) in order to increase modeling power. One of the most expressive models is the *Digraph Real-Time task model (DRT)* (Stigge et al. 2011b; Stigge 2014) which models each task using a directed graph. Timing constraints are represented as deadlines until which task activations need to finish their executions. However, increased expressiveness of the workload model leads to an increase in complexity of the associated schedulability analysis, which aims at statically proving the absence of deadline misses, assuming a given scheduler.

The analysis complexity is further dependent on the type of scheduler. The two common scheduler classes, *dynamic* and *static priority schedulers*, differ in how they pick which task to execute in cases where more than one of them is waiting for execution. For dynamic priority schedulers, the feasibility problem for the DRT model has been shown to be tractable for uniprocessor platforms (Stigge et al. 2011b). The method is based on evaluating demand-bound functions and leads to a pseudo-polynomial test. However, the problem has been shown to be strongly coNP-hard for static priority schedulers (Stigge and Yi 2012), implying that a pseudo-polynomial test cannot exist (assuming $P \neq NP$). Thus, the worst-case run-time of a schedulability test can be expected to be exponential in the task parameters.

One of the fundamental reasons for this hardness is that tasks do not have local worst cases which can be combined to a global worst case. As a consequence, exponentially many combinations of scenarios from all tasks need to be considered. Such *combinatorial explosions* are major sources of intractability for many problems in the theory of real-time systems and beyond. They introduce exponential algorithm run-times leading to poor scaling behaviour.

In this article, we show that by carefully considering the properties of tasks and their interactions, a static task priority schedulability test can be developed which runs efficiently for typical problem instances. The key insights are abstractions which allow to potentially prune large parts of the search space and efficiently guide the search for a deadline miss. In particular, we provide the following contributions:

- We present methods to significantly reduce the exponential number of relevant objects to be tested by introducing *dominance relations* on two domains relevant for the analysis (critical vertices and critical request functions).
- We introduce an iterative technique called *combinatorial abstraction refinement* in order to deal with a combinatorial explosion in the feasibility test for static priorities. We also provide a general, abstract formulation that can easily be instantiated for solving other combinatorial problems.

– We show the method's flexibility by extending it to two related settings. First, we investigate an extension from static *task* priorities to static *job-type* priorities by applying a busy-window extension technique. Second, we consider non-preemptive schedulers, enabling applicability of the result to domains beyond processor scheduling, e.g., networking.

Despite its exponential worst-case complexity, a prototype implementation of our method outperforms the state-of-the-art algorithm for dynamic priority feasibility which is pseudo-polynomial. Task sets with 50 tasks and more can be analyzed within a few seconds. This demonstrates competitiveness of our approach using domain knowledge for sophisticated optimizations.

We believe that our abstraction refinement technique may be applicable beyond the concrete problem we are solving in this article. Many problems in the theory of real-time systems are combinatorial in nature and our approach can be used as soon as certain lattice structures for hierarchical abstractions are defined.

1.1 Prior Work

The *periodic task model* (Liu and Layland 1973) represents each task with two integers for period and worst-case execution time (WCET). Deadlines are implicit, i.e., equal to periods. Efficient analysis procedures are known for dynamic priorities via the utilization bound (Liu and Layland 1973) and for static priorities via response-time analysis (Joseph and Pandya 1986). Task priorities can be assigned using the optimal Rate Monotonic scheme. More expressive models include the *Multiframe (MF)* (Mok and Chen 1997) and *Generalized Multiframe (GMF)* (Baruah et al. 1999) task models. Each task in these models cycles through a list of different *frames* with different execution times (for both MF and GMF) and different inter-release separation times and deadlines (only GMF). Feasibility tests for these models are based on demand-bound functions (Baruah et al. 1999). Static priority schedulability tests generalize response time analysis (Zuhily and Burns 2009; Takada and Sakamura 1997) or utilization bounds (Mok and Chen 1997; Han 1998; wei Kuo et al. 2003; Lu et al. 2007). These tests however are either imprecise, i.e., over-approximate, or very slow because of exponential explosion in complexity. For priority assignment, there is an optimal strategy called *Audsley's Algorithm* (Audsley 1991).

The most general model to date with a tractable feasibility problem is the *Digraph Real-Time task model (DRT)* (Stigge et al. 2011b; Stigge 2014). Each task is modeled with a directed graph in which vertices represent job releases and edges represent branches and inter-release delays. Feasibility for dynamic priority schedulers has been shown to be tractable (Stigge et al. 2011b), even for an extension with a bounded number of global timing constraints (Stigge et al. 2011a). However, similar results have been shown to be unlikely for static priorities since the problem becomes strongly coNP-hard in this case (Stigge and Yi 2012).

Periodic task models with non-preemptive schedulers have been analyzed in the application domain of CAN busses (Tindell and Burns 1994; Davis et al. 2007) with comprehensive general treatment by George et al. (1996). Extensions of the busy

window (Lehoczky 1990) have been investigated in the context of scheduling with varying job priorities by Harbour et al. (1991).

The introduction of *task automata* (Fersman et al. 2007) is taking a model checking approach. This model is based on timed automata, extended with real-time tasks. Expressiveness is so high that the associated schedulability problem is even undecidable in a few variants of the model. The decidable cases, including models for dynamic and static priority schedulers, suffer from the same scalability problem that is common to all model checking approaches.

An approximative solution is the *real-time calculus (RTC)* (Thiele et al. 2000) which uses arrival and service curves (not unlike the request functions we use) in order to describe task activations and availability of computing resources. There is a large body of research around RTC but it is inherently over-approximate.

Our abstraction refinement approach has been inspired by abstraction and refinement techniques used in verification, like *counterexample guided abstraction refinement* (Clarke et al. 2000) or *abstract interpretation* (Cousot and Cousot 1977; Gulavani et al. 2008).

2 Preliminaries

This section presents the task model with its syntax and semantics, followed by the problem description.

2.1 Task Model

We use the *digraph real-time (DRT) task model* (Stigge et al. 2011b) to describe the workload of a system. A DRT task set $\tau = \{T_1, \ldots, T_N\}$ consists of N independent tasks. A task T is represented by a *directed graph* $G(T)$ with both vertex and edge labels. The vertices $V(T) = \{v_1, \ldots, v_n\}$ of $G(T)$ represent the types of all the jobs that T can release. Each vertex v is labeled with an ordered pair $\langle e(v), d(v) \rangle$ denoting worst-case execution-time demand $e(v)$ and relative deadline $d(v)$ of the corresponding job. Both values are assumed to be non-negative integers. The edges of $G(T)$ represent the order in which jobs generated by T are released. Each edge (u, v) is labeled with a non-negative integer $p(u, v)$ denoting the minimum job inter-release separation time. In this work, we assume deadlines to be *constrained* by inter-release separation times, i.e., for each vertex u, its deadline label $d(u)$ is bounded by the minimal $p(u, v)$ for all outgoing edges (u, v).

Example 1 *Figure 1 shows an example of a DRT task.*

Semantics. An execution of task T corresponds to a potentially infinite path in $G(T)$. Each visit to a vertex along that path triggers the release of a job with parameters specified by the vertex label. The job releases are constrained by inter-release separation times specified by the edge labels. Formally, we use a 3-tuple (r, e, d) to

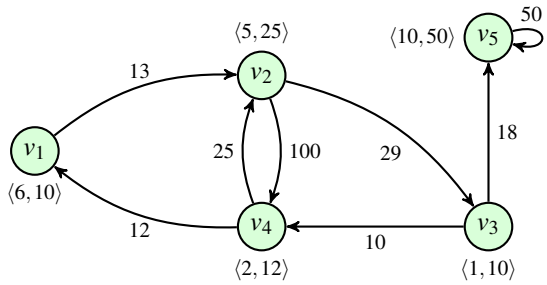


Fig. 1 An example task containing five different types of jobs

denote a *job* that is released at (absolute) time r, with execution time e and deadline at (absolute) time d. We assume dense time, i.e., $r,e,d \in \mathbb{R}_{\geqslant 0}$. A job sequence $\rho = [(r_1,e_1,d_1),(r_2,e_2,d_2),\ldots]$ is *generated by* T, if and only if there is a (potentially infinite) path $\pi = (\pi_1,\pi_2,\ldots)$ in $G(T)$ satisfying for all i:

1. $e_i \leqslant e(\pi_i)$,
2. $d_i = r_i + d(\pi_i)$,
3. $r_{i+1} - r_i \geqslant p(\pi_i,\pi_{i+1})$.

For a task set τ, a job sequence ρ is *generated by* τ, if it is a composition of sequences $\{\rho_T\}_{T\in\tau}$, which are individually generated by the tasks T of τ.

Example 2 *For the example task T in Figure 1, consider the job sequence $\rho = [(6,6,16),(19,3.3,44),(51,1,61)]$. It corresponds to path $\pi = (v_1,v_2,v_3)$ in $G(T)$ and is thus generated by T.*

Note that this example demonstrates the "sporadic" behavior allowed by the semantics of our model. While the second job in ρ (associated with v_2) is released as early as possible after the first job (v_1), the same is not true for the third job (v_3).

2.2 Schedulability and Feasibility

We assume preemptive scheduling on uniprocessor systems (except in Section 7 where we assume a non-preemptive scheduler) and use the standard notion of schedulability:

Definition 1 (Schedulability) A task set τ is *schedulable with scheduler Sch*, if and only if for all job sequences generated by τ, all jobs meet their deadlines when scheduled with *Sch*. Otherwise, τ is *unschedulable with Sch*.

While the notion of schedulability fixes a particular scheduler, feasibility is a related problem about the existence of such a scheduler:

Definition 2 (Feasibility) A task set τ is *feasible*, if and only if there is a scheduler *Sch* such that τ is schedulable with *Sch*.

We distinguish between *dynamic* and *static* priority schedulers, i.e., whether the scheduler has to obey a given order of relative priorities on the task set. In general, dynamic priority schedulers have more freedom in their scheduling decisions than static priority schedulers and can therefore successfully schedule more task sets.

For dynamic priority schedulers, it is well-known that in our setting of independent jobs, the earliest deadline first (EDF) scheduler is optimal (Dertouzos 1974). This means that if a task set can be scheduled with any scheduler, it can also be scheduled by EDF. It has been shown that for EDF, schedulability of DRT task sets can be checked in pseudo-polynomial time (Stigge et al. 2011b) for systems with bounded utilization. This is even the case for an extension of DRT with global timing constraints (Stigge et al. 2011a). Thus, feasibility is considered to be a tractable problem.

Our focus in this work is on static priority scheduling. We distinguish between static *task* priorities (SP) and static *job-type* priorities (SJP).

SP: A priority order $Pr : \tau \to \mathbb{N}$ assigns a priority to each *task* (with lower numbers for higher priorities). We assume priorities to be unique, i.e., Pr is a bijection onto $\{1, \ldots, \|\tau\|\}$.

SJP: Let $V(\tau) := \bigcup_{T \in \tau} V(T)$ denote the set of all vertices in all graphs. A priority order $Pr : V(\tau) \to \mathbb{N}$ assigns a unique priority to each *job type*.

We say that a task set τ is *SP schedulable with Pr* or *SJP schedulable with Pr* if a static task priority or static job priority scheduler using priority order Pr can successfully schedule τ, respectively. We further say that τ is *SP feasible* or *SJP feasible* if there is a Pr such that τ is SP schedulable or SJP schedulable with Pr, respectively. As we will see in Section 3, schedulability and feasibility problems are equivalent up to a linear factor. Previous work has proved that SP schedulability is strongly *coNP*-hard already for sub-classes of DRT with cyclic graphs (Stigge and Yi 2012). This means that no exact pseudo-polynomial algorithm can test SP schedulability or SP feasibility for a given task set. However, since both problems are highly relevant, we present in Sections 3 to 5 an efficient algorithm that solves typical instances in time comparable to state-of-the-art solutions for EDF schedulability.

3 Method Overview

In this section, we give an overview of our algorithm for checking SP schedulability and SP feasibility under preemptive schedulers. The method is extended to SJP schedulers in Section 6 and non-preemptive scheduling in Section 7.

3.1 Lowest-Priority Feasibility

Our decision procedures are based on checking whether a task in a task set may be assigned the lowest priority.

Definition 3 For a task set $\tau = \{T_1, \ldots, T_N\}$, a task $T \in \tau$ is *lowest-priority feasible in* τ if there is a priority order Pr with $Pr(T) = N$ such that T does not miss any deadlines if τ is SP scheduled with Pr.

Note that this definition does not state anything about deadline misses of any other tasks in τ. Further, if $T \in \tau$ is lowest-priority feasible in τ, then this property is independent of the relative priorities of all other tasks in τ. We summarize this insight in the following Lemma.

Lemma 1 *For a task set $\tau = \{T_1, \ldots, T_N\}$, if a task $T \in \tau$ is lowest-priority feasible shown by a priority order Pr, then it will always meet its deadline if SP scheduled with any permutation of Pr.*

Proof The amount of interference that a task T experiences from tasks of higher priorities does not change when their relative priorities change. In fact, even the actual interference patterns do not change, i.e., the exact timing of the interference. Thus, all permutations of priorities of tasks with higher priority than T lead to the same schedulability behavior of T. □

As we will see now, SP schedulability and SP feasibility can both be reduced to checking lowest-priority feasibility of individual tasks.

3.2 SP Schedulability

Given a task set $\tau = \{T_1, \ldots, T_N\}$ with a priority order Pr, SP schedulability of τ with Pr can be decided as follows. For each task $T \in \tau$, check whether T is lowest-priority feasible in the set of all tasks with priority up to $Pr(T)$. Note that this condition is both sufficient and necessary, since adding tasks of lower priority to a task set does neither introduce nor remove deadline misses of higher priority tasks.

Lemma 2 *A task set $\tau = \{T_1, \ldots, T_N\}$ is SP schedulable with a priority order Pr if and only if each $T \in \tau$ is lowest-priority feasible in task set $\tau_{\leqslant T}$ defined as:*

$$\tau_{\leqslant T} := \left\{ T' \mid Pr(T') \leqslant Pr(T) \right\}.$$

Proof By above discussion. □

3.3 SP Feasibility: Audsley's Algorithm

Checking SP feasibility of a task set τ is possible using a similar method which is usually called *Audsley's Algorithm* (Audsley 1991). First, check all $T \in \tau$ for lowest-priority feasibility in τ. If this check is successful for any T, recursively apply the algorithm to $\tau \setminus \{T\}$. However, if during this recursive procedure for some subset $\tau' \subseteq \tau$ no such T is found, τ is not SP feasible. This method has the additional advantage of synthesizing a priority order if the task set is found to be SP feasible, by taking the reverse order in which the tasks were found to be lowest-priority feasible.

Lemma 3 *A task set $\tau = \{T_1, \ldots, T_N\}$ is SP feasible if and only if either it is empty or there is a $T \in \tau$ which is lowest-priority feasible in τ and $\tau \setminus \{T\}$ is SP feasible.*

Proof It is clear that if this test succeeds, τ is indeed SP feasible since the synthesized priority order automatically satisfies the condition in Lemma 2 from above.

Conversely, let the test fail for some subset $\tau' \subseteq \tau$ but assume there is a priority order *Pr* for which τ is SP schedulable. Of all tasks in τ', some task $T \in \tau'$ is assigned lowest priority by *Pr*. Since τ is assumed to be SP schedulable with *Pr*, this task T will always meet all deadlines even with interference of all tasks of higher priority in τ. However, τ' is just a subset of τ, therefore the interference experienced by T from all other tasks in τ' can not be larger (Lemma 1). Thus, T must be lowest-priority feasible in τ', contradicting the assumption that the test failed for τ'. □

This lemma provides a recursive instance of Audsley's optimal priority assignment scheme. Our setting satisfies the three conditions that are sufficient for it to apply (Davis and Burns 2011). Note that this means that in the process of synthesizing a priority order starting with the lowest priority, one can never "pick wrong" among all tasks that are lowest-priority feasible.

3.4 Critical Vertices

As we will see now, it is sufficient to test all vertices of a task separately in order to conclude that the task is lowest-priority feasible. The fundamental assumption for this to hold is that deadlines are constrained, which implies that jobs of the same task do not cause interference to each other[1].

More specifically, given a vertex v with WCET $e(v)$ and relative deadline $d(v)$, it is sufficient to check whether the tasks of higher priority τ_{high} can execute for an accumulated time of strictly more than $d(v) - e(v)$ time units in any time interval of size $d(v)$. In case that is possible, the task containing v can not be lowest-priority feasible since the corresponding job may miss its deadline. However, if that is not the case, we say that v is *schedulable with interference set* τ_{high} or just *schedulable* if τ_{high} is clear from the context. Our algorithm for testing schedulability of a single vertex is described in Section 4.

Lemma 4 *Given a task set τ, a task $T \in \tau$ is lowest-priority feasible if and only if all vertices $v \in G(T)$ are schedulable with interference set $\tau \setminus \{T\}$.*

Proof By above discussion. □

In fact, not all vertices need to be checked. Consider two vertices v_1 and v_2 of a task with

$$\langle e(v_1), d(v_1) \rangle = \langle 3, 10 \rangle \text{ and } \langle e(v_2), d(v_2) \rangle = \langle 2, 20 \rangle .$$

Assume that v_1 is schedulable with some interference set τ. This immediately implies that v_2 is schedulable as well, since the execution demand of jobs corresponding to v_2 is lower and only needs to meet a deadline that is larger. We say that v_1 *dominates* v_2 and call a set of vertices in a task which are *not* dominated by others *critical*

[1] Another important condition for this is that all jobs released by the same task do have the same priority. We extend the method to SJP in Section 6, i.e., where different vertices could be assigned different static priorities.

vertices. Clearly, only critical vertices need to be checked for schedulability, which also implies schedulability of all other vertices and thus lowest-priority feasibility of the whole task.

This observation is important for run-time complexity of our analysis method. For a set of n vertices with a uniform random distribution of WCET and deadlines, the expected number of critical vertices is $O(\sqrt{n})$, dramatically reducing the run-time of a loop that tests all vertices individually for schedulability. Further, as we will see in Section 4, testing a vertex v for schedulability is in the worst case exponential in $d(v)$. Therefore, an optimization that tends to remove vertices v with large $d(v)$ has the additional benefit of avoiding the most expensive individual tests.

The concept of a domination relation between two vertices can be extended to vertices of different tasks with different priorities. Take the two vertices v_1 and v_2 from above and now assume that v_2 is part of a task with higher priority than the one containing v_1. If v_1 turns out to be schedulable, then v_2 is as well, since the set of tasks interfering with the jobs corresponding to v_2 is smaller, thus causing less interference. We summarize this concept as follows.

Definition 4 For a task set τ with priority order Pr and tasks $T, T' \in \tau$, we say that $v \in G(T)$ *dominates* $v' \in G(T')$, written $v \succcurlyeq v'$, if and only if:

1. $e(v) \geqslant e(v')$,
2. $d(v) \leqslant d(v')$ and
3. $Pr(T) \geqslant Pr(T')$.

If $T = T'$, then we call this an *intra-task dominance*, otherwise an *inter-task dominance*. A maximal set of vertices v containing no other v' with $v' \succcurlyeq v$ is called a set of *critical vertices*.

For checking SP schedulability the application of this is straightforward. Test a set of critical vertices for schedulability, directly leading to SP schedulability of the whole task set. For checking SP feasibility, the priority order is not known a priori. Thus, initially only intra-task dominance can be considered. However, each time a task T is found to be lowest-priority feasible, all inter-task dominated vertices in the remaining tasks are clearly non-critical and do not need to be tested anymore.

4 Single-Job Interference Testing

We now focus on checking whether a single job may experience sufficient interference from tasks of higher priority such that it misses its deadline. For the rest of this section, we assume that we want to check schedulability of a vertex v with label $\langle e, d \rangle$, i.e., with WCET e and deadline d. We want to check whether a given task set τ of higher priority tasks may cause more than $d - e$ time units of interference in any time window of d time units. Note that the relative priorities of all tasks in the interference set τ do not matter in this case.

A naive approach for this test could be as follows. For each task $T \in \tau$, pick a path $\pi^{(T)}$. Given the set of paths $\left\{ \pi^{(T)} \right\}_{T \in \tau}$, the *synchronous arrival sequence* can be simulated, i.e., a job sequence where all jobs take their maximal execution time,

the first job from each path $\pi^{(T)}$ is released at time 0 and all following jobs as soon as allowed by the edge labels. See Figure 2 for an example. Vertex v is schedulable if and only if for an exhaustive enumeration and combination of all such paths, each simulation turned out to detect at least e idle time units within the first d time units.

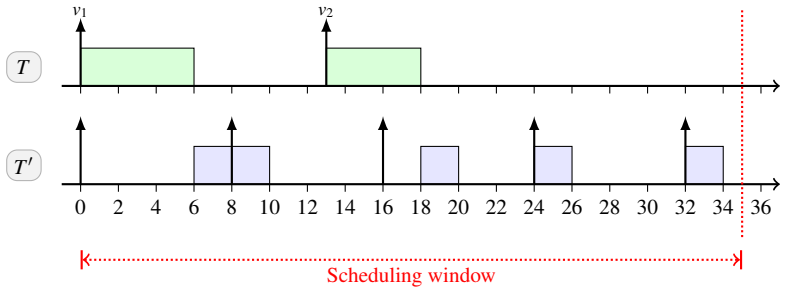


Fig. 2 Example of simulating a synchronous arrival sequence in a time interval of size 35. The interference set is $\tau = \{T, T'\}$ with T from Figure 1 and T' a periodic task with $(e,d,p) = (2,8,8)$. From task T, we simulate path (v_1, v_2, v_3). In this concrete scenario, 14 idle time units are detected.

Such an approach is of course prohibitively slow since there are two sources of exponential explosion: the number of paths in each task, and the number of path combinations to be simulated. In the rest of this section, we present ways of reducing the relevant number of paths. Section 5 introduces a method for reducing the number of combination tests.

4.1 Request Functions

In order to deal with the exponential number of paths in each task, we introduce a path abstraction that is sufficient for testing interference but allows to substantially reduce the number of paths that have to be considered. We abstract a path π with a *request function* which for each t returns the accumulated execution requirement of all jobs that π may release during the first t time units.

Definition 5 For a path $\pi = (v_0, \ldots, v_l)$ through the graph $G(T)$ of a task T, we define its *request function* as

$$rf_\pi(t) := \max\left\{e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } p(\pi') < t\right\}$$

where $e(\pi) := \sum_{i=0}^{l} e(v_i)$ and $p(\pi) := \sum_{i=0}^{l-1} p(v_i, v_{i+1})$.

In particular, $rf_\pi(0) = 0$ and $rf_\pi(1) = e(v_0)$, assuming that all edge labels are strictly positive. Note further that two paths sharing a prefix π have request functions that are identical up to the duration $p(\pi)$ of that prefix. We give an example in Figure 3.

Using this path abstraction, we can give a precise characterization of schedulability of a vertex v. The following theorem considers all combinations of all request functions corresponding to paths in all tasks of higher priority. Intuitively, the jobs

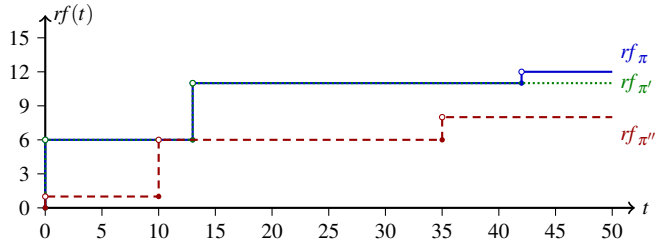


Fig. 3 Example of request functions on $[0,50]$. Paths are taken from $G(T)$ in Figure 1 with $\pi = (v_1, v_2, v_3)$, $\pi' = (v_1, v_2, v_4)$ and $\pi'' = (v_3, v_4, v_2)$. Note that π and π' share a prefix and therefore rf_π and $rf_{\pi'}$ coincide on interval $[0,42]$.

corresponding to a vertex v are schedulable if and only if for each combination there is some time interval smaller than $d(v)$ in which the sum of all requests in addition to $e(v)$ does not exceed the size of the time interval. This means that the job in question is always able to finish execution at some point before $d(v)$ time units have passed because the interference up to this point allows enough time for it to execute to completion. We write $\Pi(T)$ for the set of paths in $G(T)$ and $\Pi(\tau)$ for $\Pi(T_1) \times \ldots \times \Pi(T_N)$, i.e., the set of all combinations of paths from all tasks. Further, let $\bar{\pi} = (\pi^{(T_1)}, \ldots, \pi^{(T_N)})$ denote an element of $\Pi(\tau)$, i.e., a single combination of paths.

Theorem 1 *A vertex v is schedulable with interference set τ if and only if*

$$\forall \bar{\pi} \in \Pi(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf_{\pi^{(T)}}(t) \leqslant t. \tag{1}$$

Proof Assume Condition (1) holds but v is unschedulable. Because of the latter, there must be a combination of paths $\bar{\pi} = (\pi^{(T_1)}, \ldots, \pi^{(T_N)})$ executing in a synchronous arrival sequence for strictly more than $d(v) - e(v)$ time units within time interval $[0, d(v)]$, causing a job corresponding to v to miss its deadline. In particular, for each $t \leqslant d(v)$, tasks from τ are executing for strictly more than $t - e(v)$ time units within $[0,t]$. Since $rf_{\pi^{(T)}}(t)$ gives an upper bound for how many time units task T is executing within $[0,t]$ along path $\pi^{(T)}$, we have

$$\sum_{T \in \tau} rf_{\pi^{(T)}}(t) > t - e(v) \tag{2}$$

for each $t \leqslant d(v)$. This contradicts the assumption that Condition (1) holds.

Assume now that v is schedulable but Condition (1) does not hold. Because of the latter, there is $\bar{\pi} \in \Pi(\tau)$ such that Condition (2) holds for all $t \leqslant d(v)$. Let $t_0 \leqslant d(v)$ minimal such that τ leaves $e(v)$ time units of idle time in $[0, t_0]$ when $\bar{\pi}$ is executing in a synchronous arrival sequence. Such a t_0 must exist since v is schedulable. Thus, up to t_0, the accumulated sum of execution times of jobs released along the paths $\pi^{(T)}$ does not exceed $t_0 - e(v)$. Since there is idle time at t_0, this accumulated sum is equal to $\sum_{T \in \tau} rf_{\pi^{(T)}}(t_0)$ by Definition 5, so Condition (2) can not hold for this particular t_0, leading to a contradiction. □

Note that in Condition (1) it is sufficient to only test all integers $t \leqslant d(v)$ since request functions only change at integer points. There are two reasons for that: (1) we assume all graph labels to be integers, and (2) the release sequences represented by request functions have all job releases as early as possibly allowed by the edge labels. For the rest of the article, functions only need to be evaluated at integer points.

Generally, each $\Pi(T)$ may be infinite, since there are infinitely many paths in directed graphs with (directed) cycles. However, as we have already seen for paths sharing a prefix, only finitely many prefixes of paths in $\Pi(T)$ are relevant. This is because only a bounded number of them has request functions that differ somewhere on the interval $[0, d(v)]$. Formally, let $RF(T)$ denote the set of request functions corresponding to the paths in $G(T)$, restricted to domain $[0, d(v)]$. As with Π before, we write $RF(\tau)$ for all combinations of tasks, i.e., $RF(T_1) \times \ldots \times RF(T_N)$ and $\bar{rf} = (rf^{(T_1)}, \ldots, rf^{(T_N)})$ for elements of $RF(\tau)$. With this notation, Condition (1) is equivalent to

$$\forall \bar{rf} \in RF(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) \leqslant t. \tag{3}$$

4.2 Critical Request Functions

The test in Condition (3) is already finite, since $RF(\tau)$ can be effectively and finitely enumerated. However, the number of request functions per task is exponential in d. We will see that only a small fraction of them is in fact relevant. Consider two request functions rf and rf' such that $rf(t) \geqslant rf'(t)$ for all t in $[0, d(v)]$. If Condition (3) is satisfied using rf for some task, then it will also be satisfied with rf' instead for the same task, since the LHS of the inequality is even smaller with rf'. Clearly, only rf needs to be considered.

We formalize this by introducing a notion of dominance on the set of request functions.

Definition 6 For two request functions rf and rf' on domain $[0, d]$, we say that rf *dominates* rf', written $rf \succcurlyeq rf'$, if and only if

$$\forall t \in [0, d] : rf(t) \geqslant rf'(t).$$

A maximal set of request functions rf containing no other rf' with $rf' \succcurlyeq rf$ is called a set of *critical request functions*.

Example 3 *As an example, we take again the request functions on* $[0, 50]$ *in Figure 3. Note that* rf_π *has at each point a value at least as large as* $rf_{\pi'}$*. Therefore we have* $rf_\pi \succcurlyeq rf_{\pi'}$*. The same holds with* $rf_{\pi''}$*, i.e.,* $rf_\pi \succcurlyeq rf_{\pi''}$*. In fact,* rf_π *is a* critical *request function for this task.*

Let $CRF(T)$ denote the (unique) set of critical request functions for T and let $CRF(\tau)$ be defined analogously. Then Condition (3) is equivalent to the following which is quantifying only over all combinations $CRF(\tau)$ of *critical* request functions instead of $RF(\tau)$.

$$\forall \bar{rf} \in CRF(\tau) : \exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) \leqslant t. \tag{4}$$

Typically, only a rather small fraction of request functions in a task is critical (tens versus thousands or millions). This already reduces the number of combinations dramatically for which Condition (4) needs to be checked, despite the theoretically exponential size of all $CRF(T)$ in the worst case.

4.3 Computation of Request Functions

Critical request functions are our solution to the exponential explosion of paths in each graph $G(T)$ by carefully considering only the relevant ones. Before we present in Section 5 our solution to the second source of exponential complexity, i.e., the number of *combinations* of request functions from different tasks, we sketch an efficient method for computing critical request functions for a given graph $G(T)$.

The algorithm is based on an iterative graph exploration technique presented in (Stigge et al. 2011b) based on path abstractions, which in our case are request functions. The idea is to start with all 0-paths in the graph, i.e., paths containing just a single vertex, and iteratively extending each already generated path with all successor vertices. During that procedure, request functions that are found to be dominated by an already generated one are discarded. The procedure ends when all critical request functions on domain $[0, d(v)]$ have been generated. For more details about the general algorithm framework we refer to (Stigge et al. 2011b).

4.4 The Naïve Algorithm

We summarize this section by presenting a naïve first version of the full algorithm in Figures 4 to 7, based on Lemmas 2 to 4 and Theorem 1. Assumed is a function *generate-rfs*(T) returning a set of critical request functions for a task T on the relevant time interval. Such a function can be implemented as sketched above in Section 4.3. We further assume that vertices have been marked as critical and implicitly update these markings in *SP-feasible*(τ).

function *schedulable*(v, τ) :
1: **for all** $T \in \tau$ **do**
2: $\quad CRF(T) \leftarrow$ *generate-rfs*(T)
3: **end for**
4: **for all** $\bar{rf} \in CRF(\tau)$ **do**
5: $\quad$ **if** $\forall t \leqslant d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) > t$ **then**
6: $\quad\quad$ **return false**
7: $\quad$ **end if**
8: **end for**
9: **return true**

Fig. 4 Algorithm for schedulability of a vertex v with interference set τ.

Note that *SP-schedulable*(τ, Pr) makes $O(\|\tau\|)$ calls to *lp-feasible*(T, τ), compared to *SP-feasible*(τ) making $O(\|\tau\|^2)$ such calls. Thus, the run-time difference is only about a factor linear in the number of tasks.

function *lp-feasible*(T, τ) :
1: **for all** critical $v \in G(T)$ **do**
2: **if not** *schedulable*$(v, \tau \setminus \{T\})$ **then**
3: **return false**
4: **end if**
5: **end for**
6: **return true**

Fig. 5 Algorithm for lowest-priority feasibility of a task $T \in \tau$.

function *SP-schedulable*(τ, Pr) :
1: **for all** $T \in \tau$ **do**
2: **if not** *lp-feasible*$(T, \tau_{\leqslant T})$ **then**
3: **return false**
4: **end if**
5: **end for**
6: **return true**

Fig. 6 Algorithm for SP schedulability of a task set τ with priorities Pr.

function *SP-feasible*(τ) :
1: **if** $\tau = \emptyset$ **then**
2: **return true**
3: **end if**
4: **for all** $T \in \tau$ **do**
5: **if** *lp-feasible*(T, τ) **then**
6: **return** *SP-feasible*$(\tau \setminus \{T\})$
7: **end if**
8: **end for**
9: **return false**

Fig. 7 Algorithm for SP feasibility of a task set τ.

The main bottleneck of both algorithms is the combinatorial explosion in line 4 of *schedulable*(v, τ). Even though the number of critical request functions per task is low, a brute force style test of all combinations is still prohibitively expensive. We deal with this problem in the following section by replacing the rather naive combinatorial test with our proposed iterative approach using combinatorial abstraction refinement.

5 Combinatorial Abstraction Refinement

In the previous section we dealt with exponential problem sizes by introducing dominance relations on the domains of vertices and request functions in order to discard large fractions of the search space. However, the combinatorial problem of having to try all combinations of (critical) request functions remains.

In order to deal with this problem, we introduce an abstraction on top of request functions, called *abstract request functions*. This abstraction is still sound: if a combination of abstract request functions signals schedulability of a vertex, this conclusion is indeed true. However, if a combination signals non-schedulability, it may be that this conclusion is over-approximate. In such a case, in order to still give precise results, we *refine* the abstraction into combinations of "less abstract" request functions.

This is iterated until either a vertex is finally found to be schedulable, or we arrive at a combination of concrete request functions, i.e., without any remaining abstraction, conclusively resulting in unschedulability.

As we will see, the result is a precise analysis method which avoids combinatorial explosion for typical inputs. The rest of this section presents the details of our technique.

5.1 Abstract Request Functions

We introduce an abstraction of a *set* of request functions by taking their point-wise maximum.

Definition 7 We call rf a *concrete request function* if it is derived from a path π in a graph $G(T)$ as in Definition 5.

We call rf an *abstract request function* if there is a set $\{rf_1, \ldots, rf_k\}$ of concrete request functions, such that

$$\forall t : rf(t) = \max\{rf_1(t), \ldots, rf_k(t)\}.$$

In that case we write $rf = rf_1 \sqcup \ldots \sqcup rf_k$.

Abstract request functions can be directly used in a schedulability test in order to get over-approximate results. Specifically, for each task T, let $mrf^{(T)}$ be the abstract request function derived from the *whole* set $CRF(T)$ of all critical (concrete) request functions. We call $mrf^{(T)}$ the *most abstract request function* for T. Using the combination of all $mrf^{(T)}$, a vertex v is schedulable if

$$\exists t \leqslant d(v) : e(v) + \sum_{T \in \tau} mrf^{(T)}(t) \leqslant t. \tag{5}$$

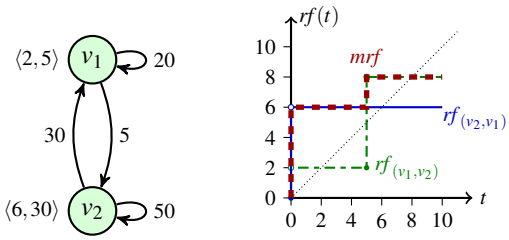
This holds because Condition (5) implies Condition (4) from Section 4.2. The test is much more efficient since it uses only *one* combination of (now abstract) request functions instead of exponentially many.

However, Condition (5) is over-approximate. If it is satisfied, vertex v is indeed schedulable, but if it fails, v may still be schedulable. See Figure 8 for an example of this. The reason is that the abstraction loses information, and therefore the implication does not hold in the other direction, i.e., Conditions (4) and (5) are *not* equivalent.

In order to turn this back into a precise test while still taking advantage of the abstraction power, we now introduce an abstraction refinement technique which allows us to iteratively refine the abstraction until a precise answer can be given.

5.2 Abstraction Refinement

As we have seen above, testing schedulability using just the most abstract request function may give an imprecise result in case the test fails. Instead of falling back to testing all combinations of concrete request functions, the abstraction can be *refined* by trying intermediate steps. For example, the test can be applied to abstract request



(a) $G(T)$ of task T (b) Request functions

Fig. 8 Example demonstrating imprecise results if just the most abstract request function *mrf* is used. The two critical request functions $rf_{(v_1,v_2)}$ and $rf_{(v_2,v_1)}$ for T are shown. Both scenarios identify idle intervals on $[0,8]$: path (v_1,v_2) from $t=2$ and path (v_2,v_1) from $t=6$. Thus, a vertex with $\langle e,d\rangle = \langle 1,8\rangle$ would be schedulable with T having higher priority. However, this information is lost if only *mrf* is considered.

functions that do not take the maximum over *all* concrete request functions, but for example just half of them. The result is a test which is more precise than Condition (5) but still more efficient than Condition (4). Since this step is more precise, it is more likely that the test succeeds in case v is schedulable. If the test still fails, the abstractions can be refined even further.

We now make this idea formal and present the details. For each task T, we build an *abstraction tree* bottom-up as follows. The leaves are represented by all concrete request functions from $CRF(T)$. In each step of the construction, we take two nodes rf_1 and rf_2 which do not yet have a parent node and which are "closest", for example by using a similarity metric on request functions (see Section 5.3). For these two nodes, we create their parent node by taking their point-wise maximum $rf_1 \sqcup rf_2$. This is repeated until we have created the full tree, in which case the tree root is the most abstract request function $mrf^{(T)}$. Figure 9 illustrates the abstraction tree[2].

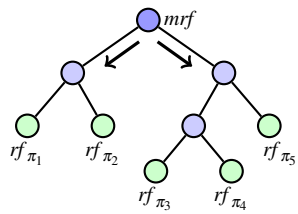


Fig. 9 Request function abstraction tree for request functions of task T in Figure 1. The leaves are all five concrete (critical) request functions on $[0,50]$. Each inner node is the point-wise maximum of all descendants and thus an abstract request function. Abstraction refinements happen downwards along the edges, starting at the root.

Our abstraction refinement algorithm works on these abstraction trees as follows. First, test schedulability of a vertex v by testing the combination of all tree roots, ex-

[2] The point-wise maximum on request functions and the dominance relation from Definition 6 are a join-semilattice $(\succcurlyeq, \sqcup)$ on the request functions for each task. These semilattices are the core structure of our abstraction refinement technique.

actly as in Condition (5). If that test fails, replace one of the abstract request functions with its child nodes from the tree, creating several new combinations to be tested. This is iterated until either all tests conclude that v is schedulable, or until a combination of leaves, i.e., concrete request functions, turns out to make v unschedulable.

The resulting method is precise and much more efficient than testing all possible combinations of request functions. The reason for the efficiency is that in the schedulable case, the test is likely to succeed already on rather high abstraction levels. Further, in the unschedulable case, the combination of concrete request functions that violates schedulability is found in a rather guided way through the trees down to the tree leaves since schedulable subtrees are avoided.

5.3 Similarity Metric

We use a similarity metric on request functions in two situations: when building the abstraction tree and when refining a combination of abstract request functions.

Building the Abstraction Tree: During construction of the abstraction tree, we want to merge the two "most similar" request functions. The effect of this is that the abstract request function representing them is a good representation of the two abstracted ones.

Abstraction Refinement: When a combination of request functions signals a potential deadline miss, we want to replace one of them with its child nodes in the corresponding abstraction tree of its task. It is beneficiary to choose the one where the child nodes are "least similar" since this will lead to rather different situations being tested next, i.e., different regions of the search space.

Formally, we define a metric on the space of request functions. It captures our intuitive notion of "distance" between two functions as representing the difference in behavior in simulated sequences.

Definition 8 For two request functions rf and rf' we define their *distance* on domain $[0,d]$ as

$$dist_d(rf, rf') := \sum_{i=0}^{d} \alpha^i \cdot |rf(i) - rf'(i)|.$$

We choose to introduce a weighting factor α which results in differences in early values weighing more than in later values. The rationale is that idle intervals early in the considered synchronous arrival sequence have an overall larger effect on schedulability of a vertex. Therefore, request functions that are very similar early in the interval should be considered more alike than request functions that are rather different early in the interval and only become more similar later. In our tests, we found that a good compromise value is when early values are weighted with a factor of about 10 compared to late values. This leads to $\alpha = \sqrt[d]{0.1}$ with $\alpha^0 = 1$ and $\alpha^d = 0.1$.

5.4 The Improved Algorithm

We give now the full algorithm that incorporates the abstraction refinement technique. The only change to the pseudo-code given in Section 4 is the implementation of *schedulable*(v,τ) which we replace with *schedulable-car*(v,τ) in Figure 10.

The implementation assumes a function *generate-mrf*(T) which generates the abstraction tree and returns the tree root, i.e., the most abstract request function for T. We further assume a function *refine*$(\bar{rf})$ which takes a combination of request functions and returns a set of combinations where one or more of the abstract request functions are replaced by child nodes from the abstraction tree(s). Further, the implementation uses a store for combinations of request functions. This could be a stack or a queue or any other data structure that implements insertion (*add*) and retrieval (*pop*) operations and a test for emptiness (*isempty*). The algorithm returns if either a combination of concrete request functions is found to make v unschedulable or if the test of all combinations concludes schedulability of v.

function *schedulable-car*(v,τ) :
1: $store \leftarrow \emptyset$
2: **for all** $T \in \tau$ **do**
3: $\quad rf^{(T)} \leftarrow$ *generate-mrf*$(T, d(v))$
4: **end for**
5: *store.add*$(\bar{rf})$
6: **while not** *store.isempty*() **do**
7: $\quad \bar{rf} \leftarrow$ *store.pop*()
8: $\quad$ **if** $\forall t \leqslant d(v) : e(v) + \sum_{T\in\tau} rf^{(T)}(t) > t$ **then**
9: $\quad\quad$ **if** *isabstract*$(\bar{rf})$ **then**
10: $\quad\quad\quad$ *store.add*(*refine*$(\bar{rf})$)
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ **return false**
13: $\quad\quad$ **end if**
14: $\quad$ **end if**
15: **end while**
16: **return true**

Fig. 10 Improved algorithm based on combinatorial abstraction refinement for schedulability of a vertex v with interference set τ.

5.5 General Algorithm Formulation

We conclude Section 5 by providing a general formulation of the refinement procedure from which the schedulability analysis above can be instantiated. Generally, the method provides an approach to finding negative instances of a combinatorial decision problem, or proving the absence thereof. The method is applicable to any N-ary predicate with abstraction structures in all components. Formally, we assume the following.

– The problem is defined on N finite domains $S_1, \ldots, S_N$. An element of the product space $S_1 \times \ldots \times S_N$ is called a *concrete combination*.

- Each domain S_i is embedded in a domain A_i of *abstract* elements, i.e., $S_i \subseteq A_i$. On each A_i there is a partial order $\succcurlyeq_i$ in which S_i are the minimal elements and there is a greatest element $\top_i \in A_i$.
- The partial orders on the component domains implicitly define a partial order $\succcurlyeq$ on the product space:

$$a_1 \succcurlyeq_1 a'_1 \wedge \ldots \wedge a_N \succcurlyeq_N a'_N \iff \langle a_1, \ldots, a_N \rangle \succcurlyeq \langle a'_1, \ldots, a'_N \rangle .$$

  This implies that all elements from $S_1 \times \ldots \times S_N$ are the minimal elements of $\succcurlyeq$ and that $\langle \top_1, \ldots, \top_N \rangle$ is the greatest element.
- There is a predicate

$$P : A_1 \times \ldots \times A_N \to \{true, false\} .$$

  For concrete combinations, this predicate distinguishes positive from negative instances.
- The predicate is monotonic with respect to the partial order, that is:

$$\left[P(\langle a_1, \ldots, a_N \rangle) \ \wedge \ \langle a_1, \ldots, a_N \rangle \succcurlyeq \langle a'_1, \ldots, a'_N \rangle \right] \implies P(\langle a'_1, \ldots, a'_N \rangle).$$

Combinatorial abstraction refinement provides an efficient method of finding a concrete combination $\langle s_1, \ldots, s_N \rangle \in S_1 \times \ldots \times S_N$ where $P(\langle s_1, \ldots, s_N \rangle)$ is *false*, or proving that none exists. Note that this problem is in *coNP* if the predicate P can be evaluated in polynomial time. It may be interpreted as proving a theorem over a domain composed of orthogonal components, or fining a counterexample to that theorem.

Except for a few special cases, a naïve brute-force method would need $\prod_i \|S_i\| = \Omega(2^N)$ evaluations of P, i.e., an exponential number, since the necessity of having to try all combinations leads to a combinatorial explosion. The refinement scheme is often much more efficient than that, depending on the abstractions. We give pseudo-code of the general formulation in Figure 11. As for its instances used for testing schedulability above, we assume a few associated auxiliary functions.

- A data structure *store* used for storing and retrieving tuples from the abstract product domain $A_1 \times \ldots \times A_N$. The store needs to support functions *add* for storing and *pop* for retrieving tuples. A function *isempty* indicates whether the store is empty.
- A function *refine* takes one tuple $\langle a_1, \ldots, a_N \rangle$ as its argument and returns a set of tuples $\{\langle b_1, \ldots, b_N \rangle_i\}$ which, intuitively, is a complete set of direct descendants in $\succcurlyeq$. Formally:
  1. Each $\langle b_1, \ldots, b_N \rangle_i$ is dominated by $\langle a_1, \ldots, a_N \rangle$, i.e.,

$$\forall i : \langle a_1, \ldots, a_N \rangle \succcurlyeq \langle b_1, \ldots, b_N \rangle_i .$$

  2. For any concrete combination $\langle s_1, \ldots, s_N \rangle \in S_1 \times \ldots \times S_N$ which is dominated by $\langle a_1, \ldots, a_N \rangle$, i.e.,

$$\langle a_1, \ldots, a_N \rangle \succcurlyeq \langle s_1, \ldots, s_N \rangle ,$$

  there is one of the new tuples $\langle b_1, \ldots, b_N \rangle_i$ dominating it, i.e.,

$$\exists i : \langle b_1, \ldots, b_N \rangle_i \succcurlyeq \langle s_1, \ldots, s_N \rangle .$$

Note that the new tuples $\langle b_1, \ldots, b_N \rangle_i$ may partially coincide with the original $\langle a_1, \ldots, a_N \rangle$. As an example, if each partial order $\succcurlyeq_i$ can be represented as a tree, *refine* may return a set of combinations for which *one* a_i in tuple $\langle a_1, \ldots, a_N \rangle$ is replaced by all child nodes in its tree.

– A function *isabstract* testing whether a tuple $\langle a_1, \ldots, a_N \rangle$ is concrete or not, i.e.,

$$\mathit{isabstract}(\langle a_1, \ldots, a_N \rangle) \iff \langle a_1, \ldots, a_N \rangle \notin S_1 \times \ldots \times S_N.$$

Clearly, the method presented above in Figure 10 is an instance of this general formulation in which $S_i = \mathit{CRF}(T_i)$, predicate P is the existence of an idle instant and all partial orders in all components are given by the abstraction trees. (Note that the return value is just the opposite since the algorithm in Figure 10 returns *true* if the vertex is schedulable while the algorithm in Figure 11 returns *false* in that case since no counter-example has been found.)

function *find-negative*(P) :

```
 1: store ← ∅
 2: store.add(⟨⊤_1, ..., ⊤_N⟩)
 3: while not store.isempty() do
 4:     ⟨a_1, ..., a_N⟩ ← store.pop()
 5:     if ¬P(⟨a_1, ..., a_N⟩) then
 6:         if isabstract(⟨a_1, ..., a_N⟩) then
 7:             store.add(refine(⟨a_1, ..., a_N⟩))
 8:         else
 9:             return ⟨a_1, ..., a_N⟩
10:         end if
11:     end if
12: end while
13: return false
```

Fig. 11 Generalized formulation of our combinatorial abstraction refinement technique. If the input problem P contains negative instances, one of them is returned. Otherwise, the procedure returns *false*.

6 Static Job-Type Priorities

In this section, we extend the above methods to static job-type priorities. That is, we assume a priority order Pr for all vertices in all graphs, but vertices of a task may have different priorities. A central observation is that the concept of lowest-priority feasibility as introduced in Section 3.1 can be defined for vertices as well. We also show that Audsley's Algorithm can be applied to this setting. However, we will see that a busy-window extension technique is necessary since tasks may suffer from indirect self-interference.

6.1 Lowest-Priority Feasibility for Vertices

We generalize lowest-priority feasibility as defined for tasks in Definition 3 to vertices, as follows.

Definition 9 For a task set τ, a vertex $v \in V(\tau)$ is *lowest-priority feasible in* τ if there is a priority order Pr with $Pr(v) = \|\tau\|$ such that jobs corresponding to v do not miss any deadlines if τ is SJP scheduled with Pr.

As for lowest-priority feasibility of *tasks*, this notion for *vertices* is independent of the relative order of vertices with higher priorities. With this property, we can reduce both SJP schedulability and SJP feasibility to lowest-priority feasibility of vertices. We provide two lemmas which are similar to the counterparts for SP from Sections 3.2 and 3.3. For a task T and a vertex $v \in V(T)$, we write $T[\leqslant v]$ for a modification of T in which all vertices v' with $Pr(v') > Pr(v)$ have their WCET parameters set to $e(v') = 0$, i.e., only vertices of higher priority than v are actually executing. Further, we write $\tau[\leqslant v]$ in which all tasks $T \in \tau$ are either replaced with $T[\leqslant v]$ or removed altogether in case all $v' \in V(T)$ are of lower priority.

Lemma 5 *A task set τ is SJP schedulable with a priority order Pr if and only if each $v \in V(\tau)$ is lowest-priority feasible in task set $\tau[\leqslant v]$.*

Proof Similar to Lemma 2. Note that a task $T[\leqslant v]$ may be essentially deactivated if all vertices of T have lower priority than v. □

The next lemma generalizes Audsley's Algorithm from tasks to job types and thereby provides a method for synthesizing a feasible priority order of job types. We write $\tau[< v]$ for τ with similar modifications as $\tau[\leqslant v]$, but excluding v itself.

Lemma 6 *A task set τ is SJP feasible if and only if either it is empty or there is a $v \in V(\tau)$ which is lowest-priority feasible in τ and $\tau[< v]$ is SJP feasible.*

Proof Similar to Lemma 3. □

In summary, the problem reduces again to solving a lowest-priority feasibility problem, this time for vertices.

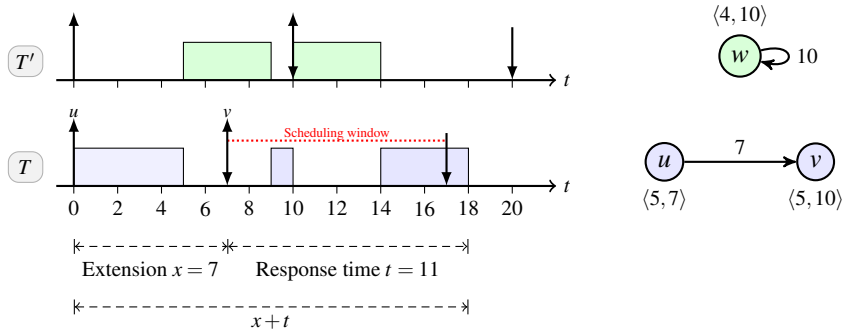
6.2 Busy-Window Extension

As before, in order to test any particular vertex v, we can pick a set of paths through all graphs and use these to simulate a synchronous arrival sequence while skipping jobs of lower priority. The framework of using request functions as path abstractions and abstraction refinement for efficiency can be applied in a straightforward way. However, it is not sufficient to only focus on the scheduling window of v, as the following example demonstrates.

Example 4 *We construct a schedule where a job has a response time that is strictly larger than the one obtained in a synchronous arrival sequence. Consider the task set shown in Figure 12(b). Assume that v is experiencing interference from task T', i.e., jobs of T' have a higher priority than v. Further, assume that u has an even higher priority, so jobs of T' do experience interference from jobs corresponding to u.*

Figure 12(a) sketches a schedule which illustrates the relative job priorities. Note that v does not finish by its deadline since it experiences a total of 6 *time units of interference from T'. This is because of* indirect *interference of u to v. In a synchronous*

arrival sequence in which T' would have started to release jobs together with v, u would be ignored and this deadline miss would not occur.



(a) A possible schedule (b) Release structures

Fig. 12 Example demonstrating that the worst-case response time may be achieved by a job sequence which is *not* an SAS but rather a busy window extension with $x = 7$. Details are discussed in Example 4.

A solution to this problem is the common *busy window extension* technique (Lehoczky 1990). The idea is as follows. The maximal interference to v is caused by an arrival sequence where all tasks $T' \neq T(v)$ synchronously release their jobs at some time point *before* the release of v. Together with jobs from $T(v)$, the processor is kept continuously busy until v is released and finally finishes. This period is called the *busy window*. It is difficult to predict the exact size of the busy window leading to the maximal interference for v, but an upper bound can be given by computing the maximal size L of any busy window[3] for τ. With this upper bound, all possibilities can be enumerated.

We now give details of this procedure for analyzing a vertex v in a task T. We consider extensions of the analyzed window, which initially is just the scheduling window of v. We extend this window by additional x time units into the past, i.e., to the left, cf. Figure 12. For each integer $x \geqslant 0$, we analyze a scenario in which all tasks $T' \neq T$ start releasing their jobs at time 0 while the job corresponding to v is released at time x and has its deadline at $x + d(v)$, i.e., its scheduling window is the interval $[x, x + d(v)]$. In addition to this job, T releases other jobs *before* time x as late as possible, corresponding to a path through $G(T)$ *ending* in v. This construction maximizes the interference experienced by v for a particular x. In Figure 12 this is the case for $x = 7$ where T is following path (u, v). In order to find the worst-case interference, all x are enumerated up to the size L of the maximal busy window. (In practice, not all x need to be enumerated as discussed as an optimization in Section 6.4 below.)

Formally, we can use request functions as before to describe the interference to v by tasks $T' \neq T$. In addition to that, we also need to have a way of describing the

[3] A bound L for the size of the maximal busy window can be easily computed by finding smallest interval size t for which $\sum_{T \in \tau} mrf^{(T)}(t) \leqslant t$.

interference of jobs from T that are released before v, since they can cause indirect interference (by delaying jobs of some T' which then interfere with v). For this purpose, we define a *suffix request function* which is defined similarly to a request function but is built "backwards" from the end of the path. This ensures that the *last vertex* of a path is always included in the workload.

Definition 10 (Suffix Request Function) For a path $\pi = (\pi_0, \ldots, \pi_l)$ in the graph $G(T)$ of a task T, we define its *suffix request function* as

$$rf_{\pi}^{sfx}(t) := \max\left\{e(\pi') \mid \pi' \text{ is suffix of } \pi \text{ and } p(\pi') \leqslant t.\right\}$$

Example 5 *We give an example in Figure 13.*

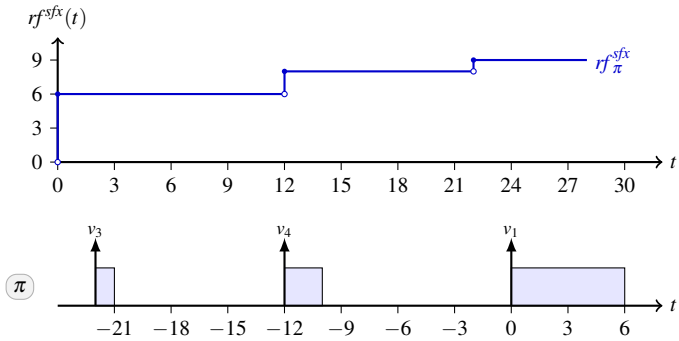


Fig. 13 Example of a suffix request function on $[0,28]$ for $\pi = (v_3, v_4, v_1)$ from $G(T)$ in Figure 1. We also show the job sequence in a graphical schedule, with time 0 being the release time of the job from v_1. This illustrates that rf_{π}^{sfx} builds *backwards* from the release of the last vertex in π.

6.3 The Full Algorithm

We can now give the full condition. As before, let $CRF(T)$ denote the set of critical request functions for a task T and let $CRF(\tau)$ denote the product set of all $CRF(T)$. Further, let $RF^{sfx}(T,v)$ denote the set of all (critical) suffix request functions for a task T which represent paths ending in $v \in V(T)$. We can express a condition for lowest-priority feasibility of a vertex v as follows, assuming T is the task containing v and τ is a task set excluding T.

$$\forall rf^{sfx} \in RF^{sfx}(T[\leqslant v], v), \bar{rf} \in CRF(\tau[\leqslant v]), x \leqslant L : \exists t \leqslant d(v) : \\ rf^{sfx}(x) + \sum_{T \in \tau} rf^{(T)}(x+t) \leqslant x + t. \quad (6)$$

This condition quantifies over all (critical) suffix request functions for T ending in v, all combinations of (critical) request functions for all other tasks, and all busy

window extensions x up to its size bound L. For all these combinations, there must be a time point t at which all workload from T, represented by term $rf^{sfx}(x)$, together with all workload from other tasks, represented by the summation term, finish. Combinatorial abstraction refinement can be applied to the first two universal quantifiers, i.e., all combinations of request functions.

Note that Condition (6) is a generalization of Condition (4) for the SP case, if we set $x = 0$. The $rf^{sfx}(x)$ term generalizes $e(v)$. For $x = 0$ we have $rf^{sfx}(x) = e(v)$, but for sufficiently large x, it also includes other vertices from T. This condition can be tested using combinatorial abstraction refinement for $\overline{rf}$ as well as rf^{sfx}, making this an efficient procedure.

6.4 Optimizations

For practical implementations, we briefly discuss two effective optimization techniques that can dramatically speed-up the analysis. The first technique is based on the observation that Condition (6) does not need to be checked for all integers $x \leqslant L$ as significant changes of the left-hand side only occur where rf^{sfx} changes. More specifically, if the inequality in Condition (6) holds for some x and t and $rf^{sfx}(x) = rf^{sfx}(x+1)$, then the inequality also holds for $x' = x+1$ and $t' = t-1$. Thus, $x+1$ does not need to be tested. This optimization reduces the domain of all x to be tested to the values at which a change in rf^{sfx} occurs. For the situation in Example 4, this is only the case for $x \in \{0, 7\}$. In general, the number of values for the size x of the busy window extension that need to be tested is related to the ratio of L to the values of all edge labels in $G(T)$. A direction worth exploring in future research is to consider applying abstraction refinement also on the domain of x itself.

Second, while the number of values for x to be tested can be dramatically reduced by the above optimization, its bound L may still be large, particularly for task sets with high utilization. The result is that request functions need to be computed for large domains. Recall that the time for computing request functions grows dramatically with larger domains. For static task priorities, this is not a problem, as the domain for request functions is restricted to scheduling windows in that case. For static job-type priorities, we need to consider busy window extensions L that can be a few times as large. An implementation of the proposed method can be optimized by computing request functions incrementally on-demand, that is, first computing the function on a prefix of its domain and only computing further values when they are needed. This optimization has the potential to avoid following branches in graphs that would lead to exponential yet unnecessary increase in analysis time.

7 Non-Preemptive Scheduling

We now change focus from preemptive to *non-preemptive* schedulers. For presentation reasons, we only describe analysis for static task priorities, but an extension to static job-type priorities is straight forward using the insights from Section 6.

In non-preemptive scheduling, a job that has started execution cannot be preempted by jobs of higher priority. This has the following consequences for the interference that needs to be taken into account in a schedulability test.

- When a job is released, the start of its execution may be delayed by interference from other jobs in the system. However, once the job starts running, no such interference is possible anymore. For our analysis, this means that instead of trying to find the latest time point where a job *finishes*, we will search for the latest time point where it can *start* executing.
- While preemptive scheduling only needs to consider interference from tasks of higher priority, tasks scheduled with a non-preemptive scheduler may experience interference from jobs of lower priority, called *blocking*. Generally, this interference is restricted to a single job of a lower-priority task, as we will see below.
- Blocking from lower-priority tasks has a similar effect as we observed with different job-type priorities: it is not sufficient to only focus on the scheduling window of a job when assessing whether it is schedulable, as illustrated in Figure 14. Thus, we need to employ the *busy window extension* technique as in Section 6.

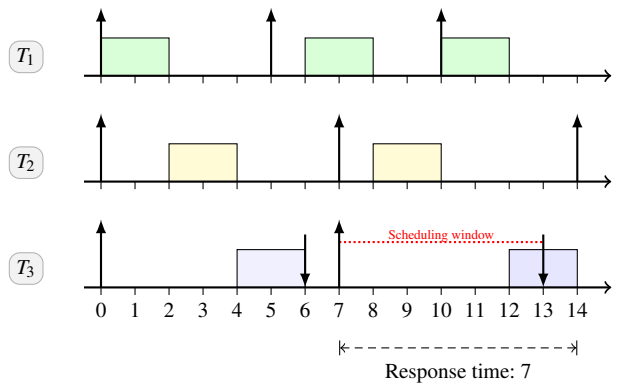


Fig. 14 Example for a non-preemptive worst-case response time which can only be detected with a busy window extension, inspired by Davis et al. (2007). Details are discussed in Example 6.

Example 6 *Consider a set of periodic tasks with constrained deadlines, i.e., each task is represented by a value e for the WCET of all its jobs, a value p for the inter-release separation delay between job releases and a value d for the relative deadline of all jobs. The task set* $\tau = \{T_1, T_2, T_3\}$ *is given as:*

Task T_1*:* $e = 2$ *and* $p = d = 5$
Task T_2*:* $e = 2$ *and* $p = d = 7$
Task T_3*:* $e = 2$ *and* $p = 7$, $d = 6$

The first job of task T_3 *finishes in time, but the second job misses its deadline, cf. Figure 14. The effect is similar to the one observed above for static job-type priorities: jobs of task* T_3 *indirectly delay jobs of the same task.*

In the rest of this section, we describe the necessary changes to the methods presented above for preemptive schedulers.

7.1 Adjusted Condition

We start by introducing the test condition for a single vertex, before presenting the bigger picture below in Section 7.2. Assume a task set τ with a priority order Pr. We want to test whether jobs corresponding to a vertex $v \in G(T)$ of a task $T \in \tau$ will always be able to meet their deadlines. Recall the busy window extension approach from Section 6. In Condition (6), for each combination $\bar{rf}$ of request functions for all tasks $T' \neq T$, each suffix request function rf^{sfx} for paths in $G(T)$ ending in v, and each busy window extension $x \leqslant L$, the condition tests whether the tested job could finish before its deadline. That is, there is a time point $t \leqslant d(v)$ such that

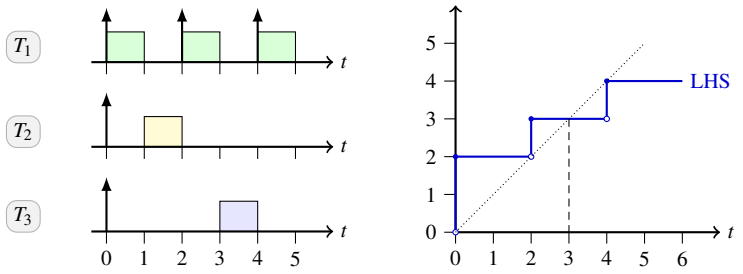
$$rf^{sfx}(x) + \sum_{T \in \tau} rf^{(T)}(x+t) \leqslant x+t.$$

This time point t represents the response time of the tested job. We will change the condition in three ways in order to reflect the behavior of non-preemptive schedulers.

1. Since the job can not be preempted anymore once it starts running, we search for its starting time instead of its finishing time. For example, in Figure 14, for the first job of T_3, we are looking for $t = 4$ where the job starts executing. This is achieved by subtracting $e(v)$ from the left-hand side of the inequality. Equivalently, one could remove $e(v)$ from rf^{sfx}, but we choose to keep notation consistent. Note that this change alone would make it possible to compute the latest start time of a job even for the preemptive setting.
2. In order for the job to be schedulable, its starting time needs to be latest $e(v)$ time units before its deadline. This condition is also sufficient, as in that case, the job is guaranteed to finish until the deadline with a non-preemptive scheduler. Therefore, we search for $t \leqslant d(v) - e(v)$.
3. Tasks of lower priority can interfere with the tested job as well, which we summarize in a blocking term $B(\tau_{>T})$. In the worst case, the largest job of all tasks of lower priority starts execution right before the first job of higher priority in the busy window is released. This means that the maximal interference caused by blocking can be given as:

$$B(\tau_{>T}) = \max\{e(v) \mid v \in G(T), T \in \tau_{>T}\}. \tag{7}$$

Finally, we need to make the request functions that are representing interference from higher-priority tasks right-continuous, in contrast to the functions used in Sections 4 to 6 which are left-continuous, cf. Definition 5. The difference is in situations where a higher-priority task releases a job at the same time as the time our test is looking for, cf. Figure 15. In previous sections, the test is deriving the response time, i.e., the time where a job finishes execution. If a higher-priority task releases a job right at that time, we do not want this event to delay the response time further. However, for



(a) Schedule for a SAS of τ

(b) LHS of Condition (8) for T_3 with $x = 0$ and $A = B = 0$

Fig. 15 Illustration for why request functions need to be *inclusive* for a non-preemptive schedulability test. Task set $\tau = \{T_1, T_2, T_3\}$ contains a periodic task T_1 with $e = 1$, $p = 2$ and two tasks T_2 and T_3 that release just one job each with $e = 1$. Clearly, T_3 can be delayed by 3 time units as shown in Figure 15(a). An exclusive request function, i.e., left-continuous, would indicate only 2 time units, cf. Figure 15(b).

non-preemptive scheduling, the test is deriving the start time of a job. If a higher-priority task releases a job at that time, we *do* want this event to delay the start time further.

Definition 11 For a path $\pi = (v_0, \ldots, v_l)$ through the graph $G(T)$ of a task T, we define its *inclusive request function* as

$$rf^{\bullet}_{\pi}(t) := \max\left\{e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } p(\pi') \leqslant t\right\}.$$

We call request functions rf_{π} from Definition 5 *exclusive request functions*. Note that the difference in the definitions of exclusive and inclusive request functions is in the different comparison relations "$<$" versus "$\leqslant$" which makes the difference between left- and right-continuous, respectively.

We use $CRF^{\bullet}(\tau)$ to denote the set of all tuples of critical inclusive request functions of all tasks. To summarize, a test for a single vertex $v \in T$ is given as

$$\forall rf^{sfx} \in RF^{sfx}(T,v), \bar{rf} \in CRF^{\bullet}(\tau_{<T}), x \leqslant L : \exists t \leqslant d(v) - e(v) : \underbrace{B(\tau_{>T})}_{A} + \underbrace{rf^{sfx}(x) - e(v)}_{B} + \underbrace{\sum_{T' \in \tau_{<T}} rf^{(T')}(x+t)}_{C} \leqslant x + t. \quad (8)$$

Note that the left-hand side is composed of three terms A, B and C, representing the blocking from lower-priority tasks, the interference from $T(v)$ itself, and the interference from higher-priority tasks, respectively.

7.2 The Full Algorithm

We now give the bigger picture. The core of our algorithm is the test of single vertices for deadline misses, given sets of tasks $\tau_{<\tau}$ and $\tau_{>\tau}$ with higher and lower priorities.

We adjust Definition 3 for lowest-priority feasibility by adding the notion of blocking in the non-preemptive setting.

Definition 12 For task sets τ and τ', a task $T \in \tau$ is *lowest-priority feasible in* τ *with blocking set* τ' if there is a priority order Pr with $Pr(T) = \|\tau\|$ such that T does not miss any deadlines if τ is SP scheduled with Pr and experiences non-preemptive blocking from τ'.

Lowest-priority feasibility of a task T in a task set τ with blocking set τ' can be tested with the method outlined above in Section 7.1 by applying Condition (8) to all vertices of $G(T)$. As in Section 3 for preemptive scheduling, we can use the lowest-priority feasibility concept to characterize SP schedulability and SP feasibility via applying it iteratively to all tasks or applying Audsley's Algorithm, respectively.

7.2.1 SP Schedulability

The only difference to the preemptive case in Section 3.2 is that we need to take the set of tasks with lower priority into account.

Lemma 7 *A task set τ is non-preemptively SP schedulable with a priority order Pr if and only if each $T \in \tau$ is lowest-priority feasible in task set $\tau_{\leqslant T}$ with blocking set $\tau_{>T}$.*

Proof By above discussion. □

7.2.2 SP Feasibility

For non-preemptive SP feasibility, we apply Audsley's Algorithm as in the preemptive case. For technical reasons, we include *blocking set* into the SP feasibility concept. On a dedicated platform with no other tasks interfering, the blocking set is empty.

Lemma 8 *A task set τ is non-preemptively SP feasible with a blocking set τ' if and only if either τ is empty or there is a $T \in \tau$ which is lowest-priority feasible in τ with blocking set τ' and $\tau \setminus \{T\}$ is SP feasible with blocking set $\tau' \cup \{T\}$.*

Proof A proof is similar to that of Lemma 3 which we will not repeat here. However, care must be taken to deal with the blocking term of Condition (8) correctly. For preemptive scheduling, there is no blocking term. Thus, a task T that is lowest-priority feasible in a task set τ is also trivially lowest-priority feasible in any subset of τ, making Audsley's Algorithm applicable. For non-preemptive scheduling, a task T that is lowest-priority feasible in a task set τ with blocking set τ' may experience more blocking if a task T' from τ (higher priority) is moved to τ' (lower priority). However, lowest-priority feasibility of T is not lost, since the potentially higher blocking due to T' if it has lower priority than T does never exceed the interference which T' can cause if it has higher priority than T. The reason is that the blocking term in Condition (8) is only given by one job, cf. Equation (7).

7.3 Comparison with Preemptive Scheduling

We briefly discuss the contrast between preemptive and non-preemptive scheduling in our setting, i.e., static priority scheduling of DRT task systems. Intuitively, a preemptive scheduler is more powerful since jobs of higher priority can get access to the processor at any time. Indeed, task sets that are non-preemptively SP feasible are commonly also preemptively SP feasible and counterexamples are rather rare. However, they do exist, as the following example demonstrates, already for periodic implicit-deadlines task sets.

Example 7 *Consider the same task set as in Example 6 but with implicit deadlines, i.e.,* $\tau = \{T_1, T_2, T_3\}$ *is given as:*

Task T_1*:* $e = 2$ *and* $p = d = 5$
Task T_2*:* $e = 2$ *and* $p = d = 7$
Task T_3*:* $e = 2$ *and* $p = d = 7$

With a preemptive SP scheduler, τ *is infeasible. Task* T_1 *can obviously not have the lowest priority. If* T_2 *or* T_3 *have the lowest priority, then the synchronous arrival sequence leads to a deadline miss since the job of lowest priority is preempted twice by* T_1*. On the other hand, with a non-preemptive SP scheduler,* τ *is schedulable if* T_1 *is assigned the highest priority, as a quick evaluation of Condition (8) reveals.*

Despite this example, a preemptive SP scheduler is generally able to schedule more task sets as we will see in Section 8. Of course, factors like preemption overheads that are not covered by our model have in practice also an impact on the relative power of both scheduling approaches. An investigation is outside the scope of this article.

8 Experimental Evaluation

We evaluate our method by running it on task sets of different sizes while measuring run-times, acceptance ratios and a set of other parameters in order to show the effectiveness of our optimization techniques. We use an implementation in the Python programming language running on a standard desktop computer. The implementation is not optimized down to the last instruction, but is suitable for a qualitative comparison of scaling properties. Task sets have typical sizes of up to 30 tasks and are analyzed within a few seconds (while a naïve enumeration approach would already take days for just five tasks). As we will see, our algorithm for SP feasibility is very effective in preventing combinatorial explosion and scales very well even though we are dealing with a *coNP*-hard problem.

8.1 Task Set Generation

Each task is generated randomly as follows. A random number of vertices is created with edges connecting them according to a specified branching degree (“fan-out”).

Edges are placed such that the graph is strongly connected. After choosing edge labels with uniform probability, deadlines are chosen randomly with a uniform ratio to the minimal outgoing edge label. Finally, execution time labels are chosen randomly with a uniform ratio to the vertices deadlines. The following table gives details of the used parameter ranges.

Vertices	Fan-out	p	d/p	e/d
$[5,10]$	$[1,3]$	$[100,300]$	$[0.5,1]$	$[0,0.07]$

Each task set is generated randomly with a given goal of a task set utilization which we define as the highest ratio of the sum of WCET vertex labels over the sum of edge labels in all cycles in $G(T)$. Tasks are added to a task set until it satisfies the given goal. Feasible task sets created by this method have sizes up to about 20 tasks with over 100 individual job types in total.

Note that exact run-times, acceptance ratios etc. may be rather sensitive to the exact way task sets are created. The evaluation in this section aims at qualitative insights, such as scaling behaviors and relative strength of acceptance ratios. A thorough study of the influence of task creation parameters is beyond the scope of this article.

8.2 Run-Time Scaling

The first property we evaluate is the scaling behavior for growing task set sizes. Recall that task set size and utilization are linearly correlated in our setting where we add tasks to a task set until a goal utilization is reached.

We compare the state-of-the-art pseudo-polynomial algorithm for EDF feasibility (Stigge 2014) with our combinatorial abstraction refinement method for SP feasibility. Recall that the latter runs in exponential time in the worst case whereas the former is pseudo-polynomial. For the resulting run-time plot in Figure 16, we analyzed about 250 task sets per slot of 3% utilization. We see that the EDF test is outperformed by the SP test until about 70% utilization after which its run-time is larger by up to 50%.

For low utilizations, the abstraction refinement method has comparable run-time and has much better scaling behavior for increasing utilizations of feasible task sets. The reason for the "bump" in the EDF curve at about 70% is caused by a phase transition between feasible and infeasible task sets (Stigge 2014). The SP test is less sensitive to phase transition effects. Its phase transition is at about 35% utilization with a slight increase in computation time, but this bump is almost not noticeable. The theoretical run-time complexity bound is exponential in the number of tasks, but the combinatorial abstraction refinement is effective in hiding the exponential growth for the analyzed tasks.

We also evaluate the SJP and non-preemptive SP feasibility tests introduced in Sections 6 and 7. We expect both to have significantly increased run-time since they need to consider analysis of the whole busy window. Figure 17 illustrates that their run-times are about an order of magnitude higher than the SP feasibility test. We

also note a clear exponential increase in run-time. This is due to several contributing factors. Higher utilization results in larger sizes of the maximal busy window. This means that the busy window extension to be considered grows with increasing utilization. The domain of request functions increases because of this, leading to increased computational effort for deriving them. Finally, the SJP feasibility analysis method has to test each job separately with changing interfering tasks and cannot benefit from re-use of request functions or optimizations like critical jobs for SP feasibility. All three factors have an exponential effect that is independent of the issue of combinatorial explosion which the abstraction refinement scheme is designed to solve.

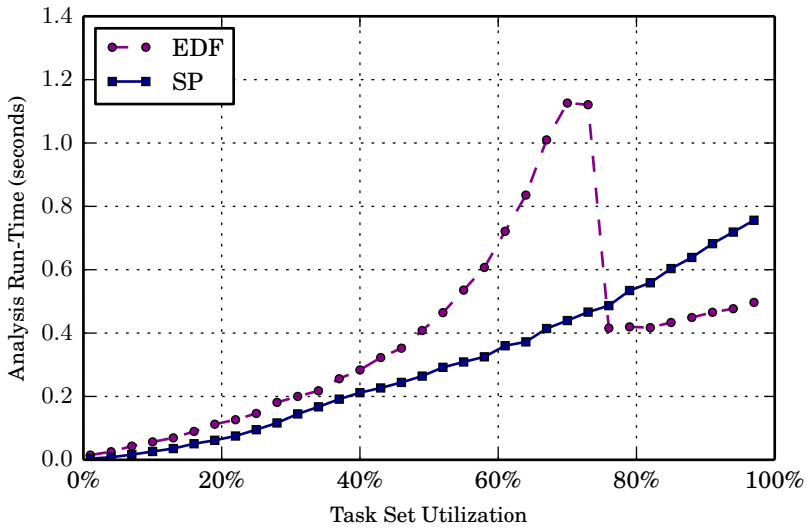


Fig. 16 Run-times of EDF and SP feasibility analyses. Our method for SP schedulability clearly outperforms the EDF test, both in absolute time and in average scaling.

8.3 Analysis Stages

We evaluate two aspects of our SP analysis method in more detail. One is the time distribution between computation of all critical request functions and checking their combinations for schedulability. The other aspect is the effectiveness of the abstraction refinement in terms of avoided combination tests.

Time Distribution Our analysis has two phases. First, all critical request functions are derived by traversing all graphs. Second, their combinations are tested using combinatorial abstraction refinement. The first phase is linear in the utilization since it is

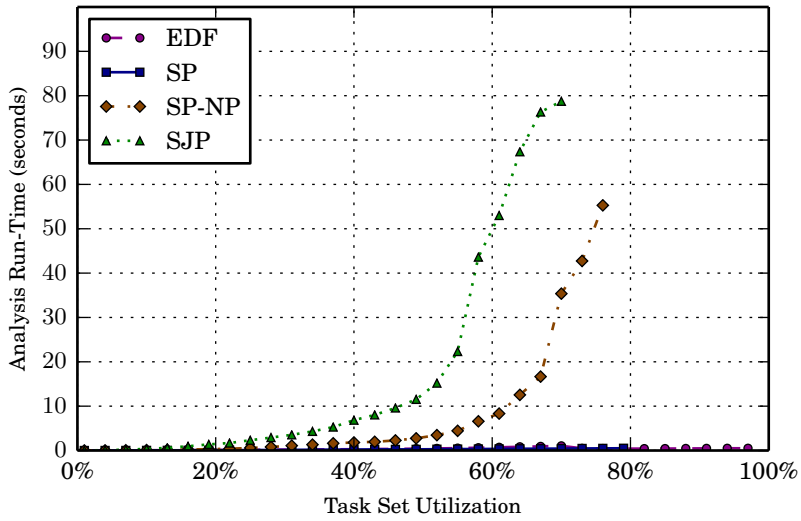


Fig. 17 Run-times of EDF, SP (preemptive and non-preemptive) and SJP feasibility analyses. Experiments were executed with a timeout of 100 seconds which occurred well beyond the non-preemptive SP and SJP feasibility phase transitions of about 35% and 55% utilization, respectively.

executed in isolation for each task, and the task set size is proportional to the utilization. However, the second phase is in the worst case exponential in the number of tasks. Therefore we expect it to grow exponentially with increasing utilization.

In Figure 18 we show the analysis run-time split into both phases. We see that the computation of request functions scales linearly as expected. The combination part grows more than linearly, but our abstraction refinement technique is very effective in dramatically reducing the combinatorial explosion. Even at a high utilization of 90% the abstraction refinement phase does not significantly exceed half of the analysis time.

Combination Testing We captured 10^5 calls to the iterative abstraction refinement procedure (Figure 10) and recorded for each call (i) how many tests where executed (Line 8 in Figure 10), (ii) how many combinations of concrete request functions there were in total and (iii) its return value. We plot the result in Figure 19 showing that our method clearly saves work in the order of several magnitudes. In more than 99.9% of all cases, less than 100 tests were executed.

8.4 Acceptance Ratio

As a last comparison, we look at acceptance ratios for EDF and static task priority (both preemptive and non-preemptive) as well as static job-type feasibility, shown

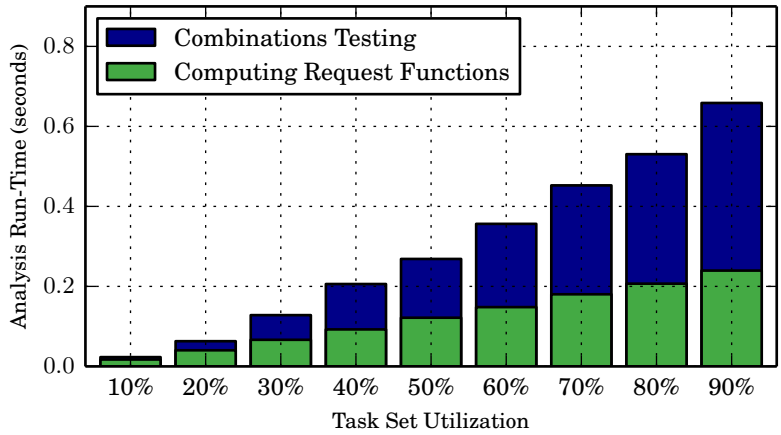


Fig. 18 Run-time of SP feasibility analysis split into computation of request functions and combination tests.

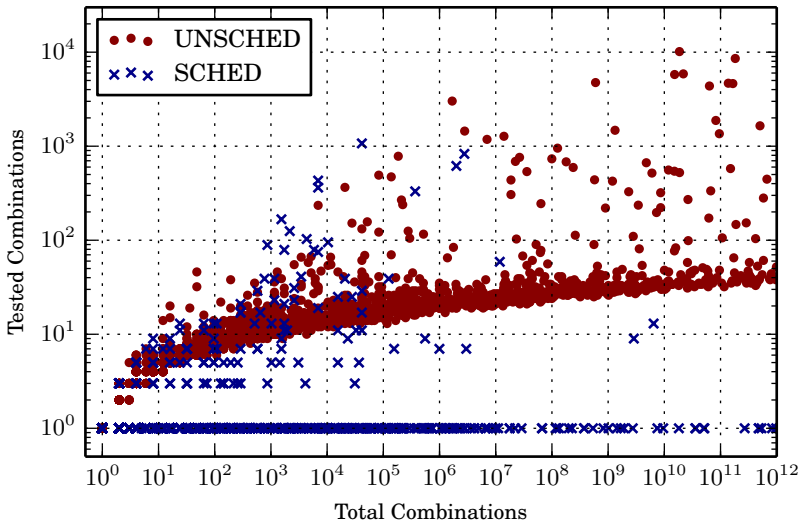


Fig. 19 Tested versus total number of combinations of request functions. Crosses represent schedulable, dots unschedulable cases. Both scales are logarithmic.

in Figure 20. Note that this comparison is *not* evaluating the quality of our analysis method. It rather compares the relative scheduling abilities of EDF, SP and SJP sheduling of DRT tasks. To the best of our knowledge, this is the first time that such a comparison can be made, since we present the first method that is able to efficiently

and precisely analyze task sets of this size for feasibility with static priority schedulers.

For the comparison of preemptive versus non-preemptive SP scheduling, we see that while preemptive scheduling appears to be more powerful than the non-preemptive counterpart, the difference is rather small. This also means that compromises between preemptive and non-preemptive schedulers, like the fixed preemption points model (Yao et al. 2010), can be assumed to not have a significant impact on acceptance ratios. However, this effect is highly dependent on the choice of task parameters, as a big job blocking a job with a short deadline from another task can easily cause deadline misses in non-preemptive scheduling. Again, a thorough investigation of task parameter generation is beyond the scope of this article.

Finally, we see that static job-type priorities do indeed significantly increase scheduling power over static task priorities, albeit at a cost of higher analysis run-times.

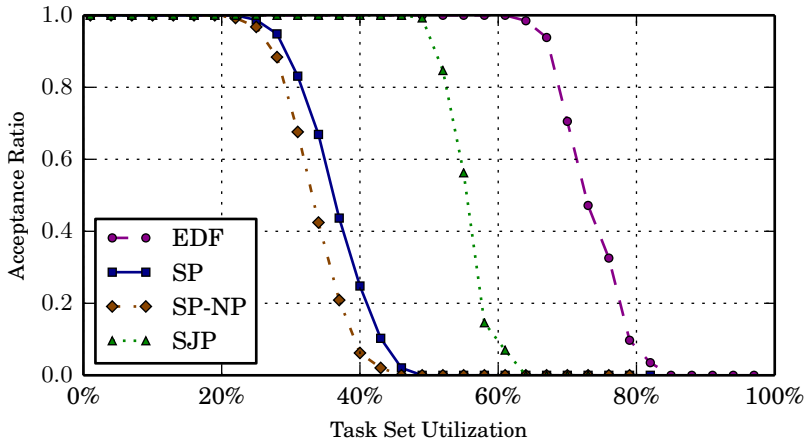


Fig. 20 Acceptance ratios of EDF and static priority schedulers.

9 Conclusions and Future Work

In this article, we have introduced an efficient method for analysing static priority feasibility and schedulability for DRT task sets. The method is based on different techniques for pruning significant parts of the worst-case exponential search space. Experiments have shown that our method has better performance than pseudo-polynomial algorithms for EDF feasibility. The extensions to non-preemptive schedulers and static job-type priorities have also been shown to perform reasonably well.

A key part of our method is the combinatorial abstraction refinement technique. Using an abstraction lattice, it allows to quickly derive results about models which

otherwise suffer from combinatorial explosion. We believe that combinatorial abstraction refinement can be applied as a general technique to many combinatorial problems that have a certain abstraction structure which is often the case for real-time scheduling problems. As future work on this concrete model, we would like to extend the specific algorithm presented in this article to variants of DRT which are extended with features like arbitrary deadlines or synchronization primitives.

A challenge appears to be the necessity of testing the whole busy window for settings like static job-type priorities or non-preemptive scheduling. One possibility for this problem is to derive conditions which guarantee that only the scheduling window needs to be tested (Yao et al. 2010). Another line of future research will be to investigate whether combinatorial abstraction refinement can be applied to busy window extensions as well, i.e., testing many extensions at once. In a broader context, we are also planning to apply the general technique to other problems in the theory of real-time systems.

Acknowledgements

The authors would like to thank the anonymous reviewers of the RTSS 2013 conference and the Real-Time Systems journal for their constructive comments on the conference version and earlier manuscripts of this article, respectively.

References

Audsley, N. C. (1991). Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS-164, University of York, England.

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

Baruah, S. K. (1998). A general model for recurring real-time tasks. In *Proc. of RTSS*, pages 114–122.

Baruah, S. K. (2010). The Non-cyclic Recurring Real-Time Task Model. In *Proc. of RTSS 2010*, pages 173–182.

Clarke, E., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In Emerson, E. and Sistla, A., editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM.

Davis, R. and Burns, A. (2011). Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. *Real-Time Systems*, 47(1):1–40.

Davis, R. I., Burns, A., Bril, R. J., and Lukkien, J. J. (2007). Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272.

Dertouzos, M. L. (1974). Control robotics: The procedural control of physical processes. In *IFIP Congress*, volume 74, pages 807–813.

Fersman, E., Krcal, P., Pettersson, P., and Yi, W. (2007). Task automata: Schedulability, decidability and undecidability. *Inf. Comput.*, 205(8):1149–1172.

George, L., Rivierre, N., and Spuri, M. (1996). Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, INRIA. Projet REFLECS.

Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. (2008). Automatically refining abstract interpretations. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Han, C.-C. J. (1998). A Better Polynomial-Time Schedulability Test for Real-Time Multiframe Tasks. In *Proc. of RTSS*, pages 104–, Washington, DC, USA. IEEE Computer Society.

Harbour, M., Klein, M., and Lehoczky, J. (1991). Fixed priority scheduling of periodic tasks with varying execution priority. In *Proc. of RTSS 1991*, pages 116–128.

Joseph, M. and Pandya, P. K. (1986). Finding Response Times in a Real-Time System. *The Computer Journal*, 29:390–395.

Lehoczky, J. (1990). Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209.

Liu, C. L. and Layland, J. W. (1973). Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61.

Lu, W.-C., Lin, K.-J., Wei, H.-W., and Shih, W.-K. (2007). New Schedulability Conditions for Real-Time Multiframe Tasks. In *Proc. of ECRTS*, pages 39–50.

Mok, A. K. and Chen, D. (1997). A Multiframe Model for Real-Time Tasks. *IEEE Trans. Softw. Eng.*, 23(10):635–645.

Stigge, M. (2014). *Real-Time Workload Models: Expressiveness vs. Analysis Efficiency*. PhD thesis, Uppsala University, Sweden.

Stigge, M., Ekberg, P., Guan, N., and Yi, W. (2011a). On the Tractability of Digraph-Based Task Models. In *Proc. of ECRTS 2011*, pages 162–171.

Stigge, M., Ekberg, P., Guan, N., and Yi, W. (2011b). The Digraph Real-Time Task Model. In *Proc. of RTAS 2011*, pages 71–80.

Stigge, M. and Yi, W. (2012). Hardness Results for Static Priority Real-Time Scheduling. In *Proc. of ECRTS 2012*, pages 189–198.

Stigge, M. and Yi, W. (2013). Combinatorial Abstraction Refinement for Feasibility Analysis. In *Proc. of RTSS 2013*, pages 340–349.

Takada, H. and Sakamura, K. (1997). Schedulability of Generalized Multiframe Task Sets under Static Priority Assignment. In *Proc. of RTCSA 1997*, pages 80–86.

Thiele, L., Chakraborty, S., and Naedele, M. (2000). Real-time calculus for scheduling hard real-time systems. In *ISCAS 2000*, volume 4.

Tindell, K. and Burns, A. (1994). Guaranteeing message latencies on control area network (can). *Proceedings of the 1st International CAN Conference*.

wei Kuo, T., pin Chang, L., hua Liu, Y., and jay Lin, K. (2003). Efficient online schedulability tests for real-time systems. *IEEE Transactions On Software Engineering*, 29:734–751.

Yao, G., Buttazzo, G., and Bertogna, M. (2010). Feasibility analysis under fixed priority scheduling with fixed preemption points. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 71–80.

Zuhily, A. and Burns, A. (2009). Exact Scheduling Analysis of Non-Accumulatively Monotonic Multi-frame Tasks. *Real-Time Systems Journal*, 43:119–146.

Authors

Martin Stigge received a Ph.D. degree in 2014 from Uppsala University for his work in the area of formal analysis of timed systems with special focus on scheduling theory. In the group of Prof. Wang Yi, he extended the theory of expressive workload models with new models, complexity results and analysis algorithms.

Wang Yi received his Ph.D. in Computer Science in 1991 from Chalmers University of Technology. Currently he is a professor at Uppsala University where he holds the chair of Embedded Systems. He is one of the initial contributors to the research area on verification of timed systems. He is a co-founder of UPPAAL, a model checker for concurrent and real-time systems. He received the CAV Award 2013, for the development of UPPAAL. His current interests include models, algorithms and software tools for modeling and verification, timing analysis, real-time scheduling, and their application to the design of embedded systems. With Pontus Ekberg, Nan Guan and Martin Stigge, he received the Outstanding paper award of ECRTS 2012 and Best Paper Awards of RTSS 2009 and DATE 2013. Wang has been an editor for several journals including IEEE Transactions on Computers and served regularly as TPC chair and TPC member for numerous conferences including TACAS, EMSOFT and RTSS. Currently he is steering committee member of EMSOFT, LCTES and ESWEEK. He is a fellow of the IEEE.