Martin Stigge

Real-Time Workload Models: Expressiveness vs. Analysis Efficiency
Dissertation presented at Uppsala University to be publicly examined in 2446, Polacksbacken, Lägerhyddsvägen 2, Uppsala, Friday, 11 April 2014 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Sanjoy Baruah (University of North Carolina at Chapel Hill).

Abstract

The requirements for real-time systems in safety-critical applications typically contain strict timing constraints. The design of such a system must be subject to extensive validation to guarantee that critical timing constraints will never be violated while the system operates. A mathematically rigorous technique to do so is to perform a schedulability analysis for formally verifying models of the computational workload. Different workload models allow to describe task activations at different levels of expressiveness, ranging from traditional periodic models to sophisticated graph-based ones. An inherent conflict arises between the expressiveness and analysis efficiency of task models. The more expressive a task model is, the more accurately it can describe a system design, reducing over-approximations and thus minimizing wasteful over-provisioning of system resources. However, more expressiveness implies higher computational complexity of corresponding analysis methods. Consequently, an ideal model provides the highest possible expressiveness for which efficient exact analysis methods exist.

This thesis investigates the trade-off between expressiveness and analysis efficiency. A new digraph-based task model is introduced, which generalizes all previously proposed models that can be analyzed in pseudo-polynomial time without using any analysis-specific over-approximations. We develop methods allowing to efficiently analyze variants of the model despite their strictly increased expressiveness. A key contribution is the notion of path abstraction which enables efficient graph traversal algorithms. We demonstrate tractability borderlines for different classes of schedulers, namely static priority and earliest-deadline first schedulers, by establishing hardness results. These hardness proofs provide insights about the inherent complexity of developing efficient analysis methods and indicate fundamental difficulties of the considered schedulability problems. Finally, we develop a novel abstraction refinement scheme to cope with combinatorial explosion and apply it to schedulability and response-time analysis problems. All methods presented in this thesis are extensively evaluated, demonstrating practical applicability.

Keywords: Real-time systems, task models, EDF, fixed-priority scheduling, schedulability analysis, response-time analysis, abstraction refinement

Martin Stigge, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Martin Stigge 2014

This work is licensed under Creative Commons Attribution-ShareAlike 4.0.

Printing of this thesis has been climate-compensated.

ISSN 1104-2516
ISBN 978-91-554-8888-8
urn:nbn:se:uu:diva-219307 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-219307)
Contents

1 Introduction ................................................................. 9
   1.1 Technical Background ............................................. 10
   1.2 Contributions ....................................................... 13
   1.3 Outline ................................................................. 14

2 A Graph-based Task Model ........................................... 15
   2.1 Job Model ............................................................ 15
   2.2 Task Model .......................................................... 16
      2.2.1 Syntax .......................................................... 16
      2.2.2 Semantics ...................................................... 18
   2.3 Schedulers and System Model .................................. 19
   2.4 Schedulability and Feasibility .................................. 20

3 Feasibility Analysis .................................................... 23
   3.1 The Feasibility Theorem .......................................... 23
      3.1.1 Demand-Bound Functions ................................. 23
      3.1.2 Feasibility Characterization ............................... 24
   3.2 An Iterative Solution .............................................. 26
      3.2.1 Demand Pairs as Path Abstractions ..................... 26
      3.2.2 Iterative Procedure using Demand Triples .......... 27
      3.2.3 Calculating the Bound ..................................... 31
      3.2.4 Computing the Utilization ................................. 36
      3.2.5 Optimizations ................................................ 40
   3.3 Arbitrary Deadlines .............................................. 42

4 Tractability Borderline ............................................... 47
   4.1 Global Constraints ................................................ 47
      4.1.1 Syntax ........................................................ 47
      4.1.2 Semantics ..................................................... 48
   4.2 Analysis for $k$-Bounded Case ................................. 49
      4.2.1 Task Transformation Procedure .......................... 51
      4.2.2 Correctness .................................................. 54
      4.2.3 Optimizations ............................................... 57
   4.3 Hardness for Unbounded Case .................................. 59
      4.3.1 Reduction Framework ....................................... 59
      4.3.2 Reduction from Hamiltonian Path Problem .......... 60
   4.4 Tractability for Subclasses ..................................... 63
      4.4.1 Hardness for E-RB and E-ncGMF ......................... 64
      4.4.2 Pseudo-polynomial Test for E-GMF ...................... 65
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.3 Analysis Stages</td>
<td>124</td>
</tr>
<tr>
<td>8.3.1 EDF Schedulability</td>
<td>124</td>
</tr>
<tr>
<td>8.3.2 SP Feasibility</td>
<td>125</td>
</tr>
<tr>
<td>8.4 Refinement Effectiveness</td>
<td>125</td>
</tr>
<tr>
<td>8.4.1 Work Reduction</td>
<td>126</td>
</tr>
<tr>
<td>8.4.2 Precision Improvement</td>
<td>126</td>
</tr>
<tr>
<td>8.5 Acceptance Ratio</td>
<td>128</td>
</tr>
<tr>
<td>9 Related Workload Models: A Survey</td>
<td>129</td>
</tr>
<tr>
<td>9.1 Model Hierarchy</td>
<td>129</td>
</tr>
<tr>
<td>9.2 From Liu and Layland Tasks to GMF</td>
<td>131</td>
</tr>
<tr>
<td>9.2.1 Periodic and Sporadic Tasks</td>
<td>131</td>
</tr>
<tr>
<td>9.2.2 The Multiframe Model</td>
<td>133</td>
</tr>
<tr>
<td>9.2.3 Generalized Multiframe Tasks</td>
<td>135</td>
</tr>
<tr>
<td>9.2.4 Non-cyclic GMF</td>
<td>138</td>
</tr>
<tr>
<td>9.3 Graph-oriented Models</td>
<td>141</td>
</tr>
<tr>
<td>9.3.1 Recurring Branching Tasks</td>
<td>141</td>
</tr>
<tr>
<td>9.3.2 Recurring Real-Time Tasks – DAG Structures</td>
<td>143</td>
</tr>
<tr>
<td>9.3.3 Non-cyclic RRT</td>
<td>146</td>
</tr>
<tr>
<td>9.4 Beyond DRT</td>
<td>147</td>
</tr>
<tr>
<td>9.4.1 Task Automata</td>
<td>147</td>
</tr>
<tr>
<td>9.4.2 Real-Time Calculus</td>
<td>148</td>
</tr>
<tr>
<td>9.4.3 Fork-Join Real-Time Tasks</td>
<td>150</td>
</tr>
<tr>
<td>10 Conclusions and Future Work</td>
<td>153</td>
</tr>
</tbody>
</table>

Acknowledgements ........................................ 157

My Publications ........................................... 159

Sammanfattning på Svenska ................................. 163

Zusammenfassung auf Deutsch .............................. 165

Index .......................................................... 167

References ..................................................... 169
1. Introduction

Recent decades have brought incredible dependency on computer systems to our society. Unprecedented in our civilization’s history, many aspects of our daily lives rely on technology with a degree of complexity that goes beyond of what any single human being can comprehend entirely. In fact, most computer systems are embedded or even hidden in other electrical devices or larger machines to provide control or monitoring services, therefore being called embedded systems. As an example, take a modern car, containing often more than a hundred embedded electronic control units. These implement many different functionalities, from critical control tasks like fuel injection or airbag inflation, to comfort-related ones like navigation systems or electric window lifts.

A common feature of embedded systems is that their timing behavior is often as important as their functional behavior. That is, a late result of a computation may be as bad as or even worse than a wrong one. This type of embedded systems is also called real-time systems. For example, a too long delay in a system controlling airbag inflation can cause the loss of lives. A more elaborate example is a safety system of a space rocket which constantly monitors flight parameters to determine whether all systems are operating as planned during the launch phase. If the conclusion that everything is normal is delivered too late, then a timeout might have already triggered an automatic self-destruction mechanism to prevent larger catastrophes.

A large fraction of embedded systems is critical from a safety point of view, making it imperative to minimize errors and failures. As system designs are usually rather complex, it is very difficult to rule out design and implementation errors just by being careful during the development process. In the end, computer systems are designed by humans, and humans make mistakes. Advanced validation and verification techniques need to be developed to achieve a high degree of certainty regarding correctness. A widely used approach is to test the implementation as thoroughly as possible within certain budgets of time and financial resources. Unfortunately, exhaustive testing is usually prohibitive since too many test scenarios would need to be considered. Therefore, other approaches which carefully analyze designs and implementations with the help of mathematical methods are attracting an increasing amount of interest from industry and academia.

This thesis is concerned with a particular type of analysis which uses mathematical rigor to prove the absence of timing errors in the design of real-time systems. The basic approach is to take a mathematical description of a system and apply analysis algorithms to prove their correctness.
1.1 Technical Background

The structure of software operating in a real-time system is typically a collection of computational tasks, sharing a processor. There are different reasons for trying to separate tasks into rather independent units of execution. First, this approach follows the divide-and-conquer design pattern which suggests to split complex problems into smaller and easier ones and to solve them in isolation. Second, the reduced complexity of each task eases the challenge of analyzing the global system behavior.

In order to analyze a system with mathematical tools, a model on an appropriate level of abstraction must be derived.

Models

Finding the right level of abstraction is a difficult problem. A model derived from a system needs to retain all details necessary to prove the desired properties and it may not be easy to know which details need to be kept. In the words of mathematician and philosopher Norbert Wiener, "The best model of a cat is another, or preferably the same, cat." However, a too high degree of detail may hide essential behavior and prevent efficient analysis procedures.

For analyzing timing properties, task models provide a good compromise. In a task model, each computational task is independently described by a set of parameters that abstract away most functional behavior and mainly focus on timing. In particular, each task is modeled as a thread of execution which executes pieces of code, called jobs, at certain points in time. One parameter associated with each job is a bound on the maximal amount of time it is expected to run on the computational resource until it is finished, called the worst-case execution time (WCET). The WCET is generally uncomputable and the estimation of tight upper bounds is a research area on its own. The results in this thesis assume that WCET bounds for the input models have been derived by appropriate means and are therefore known. Other properties of a job include its deadline, i.e., the time point at which the job must have completed its execution such that the system can meet its overall timing requirements, and release-separation delays, giving bounds on the time that passes between invocations of subsequent jobs of the same task.

Task models provide abstractions of the workload that a system encounters at run-time. Since many tasks share the same platform, they may compete for the computational resource at certain points in time. A scheduler acts as a kind of supervisor, deciding the order in which tasks are executed on the processor. An appropriate model of the scheduler is necessary to incorporate its behavior into the formal analysis. Typically, a scheduler is acting upon a few very basic principles and can therefore be modeled rather compactly. This thesis focuses on two main classes of schedulers based on static or dynamic priorities of tasks.
Analysis Properties

In order to provide proofs of correctness, the desired criteria need to be formalized as well. The most fundamental timing property is that computational jobs should be completed before their specified deadlines. That is, during the design phase, deadlines are specified by associating certain pieces of code with a time span during which they have to deliver results. For a given workload model, together with a scheduler description, the property that all tasks meet their deadlines is called schedulability. In some cases, the scheduler is not known or not yet specified, leading to the related question of feasibility, i.e., whether any scheduler could indeed manage to arbitrate access to the processor in a way such that all tasks meet their deadlines.

This thesis targets mainly the two properties schedulability and feasibility with analysis methods that are exact, i.e., not over-approximate. Both properties are qualitative in nature, that is, they are satisfied or not satisfied. One result presented in this thesis deals with a third, quantitative property called response time. It describes the time that passes from the invocation of a computational job until its completion. Thus, it does not only give black-and-white type of answers whether deadlines are met, but also quantifies the time until jobs are finished. Response-time analysis gives important insights about responsiveness of system functions necessary for reasoning about their stable functioning.

Expressiveness and Efficiency

Task models together with their analysis methods are facing a difficult trade-off between expressiveness and analysis efficiency. On one hand, models should be sufficiently expressive to enable modeling of a system’s behavior as precisely as possible. This contributes to easier modeling without restrictions and to reducing wasteful resource over-provisioning. On the other hand, the analysis of models should be efficient and scale well with the system’s size to provide results within a reasonable time frame. The main focus of this thesis is the inherent conflict between expressiveness and analysis efficiency of models. When measuring algorithm efficiency in terms of asymptotic behavior, pseudo-polynomial growth is generally considered acceptable, i.e., if the run-time of an analysis method grows in the long run according to a polynomial in model size and parameters.

Historically, the first task model with efficient schedulability tests was a very simple one. The periodic task model by Liu and Layland [30] is based on the observation that many control systems implement a periodically recurring loop with more or less the same code in each iteration. Consequently, the proposed task model characterizes each task with just two parameters which are its period and the WCET of each invocation. However, this model is very restrictive regarding expressiveness and fails to describe systems that demonstrate non-periodic behaviors. Increasingly expressive models have been pro-
posed over the years\(^1\), incorporating different types of frames, branching structures, loops, etc. Figure 1.1 provides an overview of task models, sorted by their expressiveness which negatively correlates with the analysis difficulty.

![Figure 1.1](image)

Figure 1.1. A hierarchy of task models. This thesis contributes with new models and corresponding analysis methods at the top end of the hierarchy by introducing DRT and EDRT models.

First generalizations of the periodic Liu & Layland task model include the *sporadic* task model [33], decoupling deadline from period, and the *multiframe* task model [35], introducing a set of different job types (“frames”) through which the task cycles. Both models were later unified in the *generalized multiframe (GMF)* model [12]. Despite these generalizations, efficient schedulability and feasibility analyses have been shown to exist. The model was further generalized in the *recurring branching (RB)* [8] and *recurring real-time (RRT)* [9] task models by relaxing the “linear” order in which different job types are released by a task. Both RB and RRT allow *branching code* to be modeled by a tree or directed acyclic graph, respectively, thereby greatly improving modeling expressiveness. The *non-cyclic GMF* model, proposed in [46], generalizes GMF in another direction. Here, jobs may be released in any order, therefore the behavior is not necessarily cyclic. An attempt to unify divergent generalizations has been made with the introduction of the *non-cyclic RRT* model [11]. It adds non-cyclic behavior to RRT by allowing “restarts” of the DAG traversal to be dependent on the last job released.

\(^1\)A comprehensive survey of related workload models is presented in Chapter 9 of this thesis.
A question that has been open in the real-time systems research community is: How far can task models be further generalized while efficient analysis methods are still possible?

1.2 Contributions
This thesis introduces a new graph-based task model called the Digraph Real-Time (DRT) task model. It is based on arbitrary directed graphs for job releases, strictly generalizing previously proposed models. Vertices represent computational jobs with execution time bounds and deadline information. Control flow is expressed with directed edges, labeled with separation times between job releases. By allowing arbitrary directed graphs for describing task behavior, the model enables expressing local loops, modes, etc. which is not possible in earlier, less expressive models. We show that it is the most expressive model with an exact\(^2\) pseudo-polynomial time feasibility test by presenting an iterative algorithm for analyzing preemptive uniprocessor feasibility. It is based on a novel path-abstraction framework for traversing task graphs. Several crucial optimizations are discussed for efficient implementations.

We demonstrate that the DRT task model is close to a tractability borderline. This is achieved by extending DRT tasks with global timing constraints, resulting in the Extended DRT (EDRT) model. A bounded version \(k\)-EDRT which bounds the number of global constraints by a constant \(k\) is shown to be analyzable in pseudo-polynomial time by providing a translation to basic DRT tasks. For an unbounded number of global constraints, the feasibility problem is proven strongly coNP-hard, identifying a precise tractability borderline between both model variants.

For the case of static priority schedulers, we show that the schedulability problem is strongly coNP-hard for DRT models. The hardness proof is established already for subclasses of the DRT formalism like tree and multi-frame models. We obtain the result by a sophisticated reduction construction from a partition problem, refuting previous conjectures that the problem can be solved in pseudo-polynomial time. This demonstrates the fundamental hardness of analyzing static priority scheduling in contrast to its dynamic priority counterpart.

The hardness is due to an inherent combinatorial explosion caused by combining different behaviors of the participating tasks in lack of task-local worst cases. We propose a novel abstraction refinement approach to efficiently cope with this combinatorial explosion. In combination with other techniques it significantly reduces exponential growth of run-time in the analysis. Extensive experimental results demonstrate that the refinement scheme even outperforms the pseudo-polynomial time feasibility test. The abstraction refinement

\[^2\text{With } exact \text{ we mean: not approximate, that is, the test correctly identifies feasible as well as infeasible inputs.}\]
technique is a general method and likely to be applicable to a variety of combinatorial problems in the theory of real-time systems. As another example, we apply it to response-time analysis of DRT tasks. An exact algorithm is developed using novel response-time characterizations for both static-priority and EDF schedulers. Experimental evaluations certify applicability to typical problem sizes.

1.3 Outline

The contributions are structured in the following way.

- Chapter 2 introduces the new task model and related concepts. A formal definition of precise semantics is presented and basic terminology introduced.

- A feasibility analysis method of pseudo-polynomial complexity is introduced in Chapter 3, focusing on the novel path abstraction framework for efficient graph traversal.

- The DRT workload model is extended in Chapter 4 with global inter-release separation constraints, leading to the Extended DRT (EDRT) model, which is used to establish a tractability borderline for models with a pseudo-polynomial feasibility test. Such a test is developed for a bounded version of EDRT and a hardness proof is presented for the unbounded version.

- Static priority schedulability is shown to be an intractable problem for DRT task models in Chapter 5. This result is shown to already hold for subclasses of DRT further down in the model hierarchy of Figure 1.1 on page 12. The tractability borderline is shown to be located as far down as below the GMF and multiframe models.

- An iterative refinement method called Combinatorial Abstraction Refinement is introduced in Chapter 6 to efficiently solve typical instances of the static priority schedulability and feasibility problems. The method is applied to response-time analysis of static priority and EDF schedulers in Chapter 7 and evaluated together with all other algorithms in this thesis in Chapter 8.

- Finally, Chapter 9 presents a comprehensive survey of workload models in the literature, based on the terminology and notation developed in all preceding chapters. Related work is organized in a model hierarchy for which a formal notion of model generalization is introduced.

A comprehensive list of my publications is presented in the end of this thesis, together with an overview of additional material contained herein that has not been published previously.
2. A Graph-based Task Model

In this chapter we propose a new task model based on directed graphs for describing real-time workload. All technical results in the following chapters are based on this model. We also review standard concepts from scheduling theory like different types of schedulers and notions of schedulability and feasibility that will be used in the presentation.

2.1 Job Model

A job is an abstraction of a sequential piece of code executed on the processor. Certain aspects of the code are not (directly) relevant to the analysis goals considered in this thesis, like for example functional aspects, code size or memory footprint. Instead, we are only interested its timing behavior. Therefore, a job is an abstraction with just two values: its release time, which is the absolute time point at which the code is ready to execute and can be scheduled for execution, and its execution time, recording the total amount of accumulated time for which this piece of code will occupy the computational resource until its completion. Further, a job has a job type which allows us to recognize which piece of code it abstracts, in order to statically derive properties like deadlines or control flow.

Formally, we represent a job as a triple.

Definition 2.1.1 (Job). A job \( J = (R, e, v) \) is a triple consisting of an absolute release time \( R \in \mathbb{R} \), an execution time \( e \in \mathbb{R}_{\geq 0} \) and a job type \( v \). □

For a job \( J = (R, e, v) \), its job type \( v \) is statically known. However, the values for \( R \) and \( e \) are dynamic information. They are revealed at run-time since they are part of the non-deterministic behavior of a system: job release times may be subject to external events experienced by the system; job execution times may differ depending on the hardware state and inputs to the code.

Scheduling Window

The static information about a job \( J = (R, e, v) \) includes its relative deadline \( d(v) \). This means that \( d(v) \) time units after the release of job \( J \), i.e., at time

\[^{1}\text{The execution time parameter of a job may also be interpreted as a resource budget, enforced by a monitoring process or similar.}\]
The code abstracted by \( J \) must have been executed for an accumulated time of \( e \) time units on the computational resource. Otherwise, \( J \) would miss its deadline. The time interval \([R, R + d(v)]\) is called the scheduling window of \( J \), illustrated in Figure 2.1.

![Figure 2.1. Illustration of a scheduling window.](image)

### 2.2 Task Model

The structure used to model releases of jobs is a task, abstracting a single computation thread. Tasks release jobs at certain points in time with delays in between. Traditionally, real-time scheduling theory describes a system as a collection of periodic tasks, i.e., each task releases jobs in periodic intervals. Further, all jobs released by the same task share certain properties, like bounds for their execution times or relative deadlines, i.e., they are of the same job type. However, modern real-time systems are becoming more and more complex and are not necessarily always behaving periodically. Their control structures may for example include conditional branches which in turn may contain local loops, potentially leading to complex behavioral patterns.

Therefore, the periodic task model is often not expressive enough to describe task behavior adequately. We introduce a task model which represents job releases using a directed graph for each task, modeling the control flow. This model unifies and generalizes existing models proposed and intensively studied in the literature for which a survey is provided in Chapter 9.

#### 2.2.1 Syntax

A digraph real-time (DRT) task system is a set \( \tau = \{T_1, \ldots, T_N\} \) consisting of \( N \) independent tasks. A task \( T \) is represented by a directed graph \( G(T) = (V(T), E(T)) \) with \( V(T) \) denoting its set of vertices and \( E(T) \) its set of edges.

- The vertices \( V(T) = \{v_1, \ldots, v_n\} \) represent the types of all the jobs that \( T \) can release. Each vertex \( v \) is labeled with an ordered pair \( (e(v), d(v)) \) denoting worst-case execution-time (WCET) demand \( e(v) \in \mathbb{N} \) and relative deadline \( d(v) \in \mathbb{N} \) of any corresponding job. We sometimes need to express which task a certain vertex \( v \) belongs to and use \( T(v) \) for that purpose.
The edges of $G(T)$ represent the order in which jobs generated by $T$ are released. Each edge $(u, v) \in E(T)$ is labeled with $p(u, v) \in \mathbb{N}$ denoting the minimum job inter-release separation time.

An example of a DRT task system is given in Figure 2.2.

**Figure 2.2.** An example task containing five different types of jobs.

We will implicitly assume the relation $e(v) \leq d(v)$ for all job types $v$ since we are concerned with schedulability, i.e., tasks meeting their deadlines. A job with an execution time bound that exceeds its deadline can never be guaranteed to meet the deadline, making such models uninteresting for analysis. Further, we may assume that vertex and edge labels are non-zero in all contexts concerning analysis. Simple equivalent task transformations can remove zero labels which therefore do not create any technical difficulties.

We categorize deadlines depending on their relation to labels of outgoing edges. Specifically, deadlines of a task can be any of the following.

**Constrained.** Deadlines are constrained if they are bounded by the inter-release separation times to the successor vertices, i.e., for each vertex $u$ we have

$$d(u) \leq p(u, v) \quad \text{for all edges } (u, v) \in E(T).$$

**Implicit.** Implicit deadlines are the maximal possible constrained deadlines and therefore implicitly given by the labels of the outgoing edges, i.e., for each vertex $u$ we have

$$d(u) = \min \{ p(u, v) \mid (u, v) \in E(T) \}.$$

**Monotonic absolute [12].** Deadlines are monotonic absolute if two jobs that are released from a task in a certain order also are guaranteed to have their deadlines in the same order. This is implied by constrained deadlines, but can be relaxed to having for each vertex $u$ the bound

$$d(u) \leq p(u, v) + d(v) \quad \text{for all edges } (u, v) \in E(T).$$
The literature abbreviates this property with l-MAD for “localized Monotonic Absolute Deadlines”.

**Arbitrary.** Arbitrary deadlines do not have any particular relation to the edge labels, in contrast to all other cases above. They can therefore take any value.

Clearly, implicit deadlines are a special case of constrained deadlines, which in turn are a special case of arbitrary deadlines. Less restrictions for potential deadlines of job types usually lead to more challenging analysis problems.

### 2.2.2 Semantics

We define the semantics of a DRT task system as the set of *job sequences* it may generate. This matches the intuitive goal of using tasks as models for threads with different execution traces. A job sequence models such an execution trace with all its jobs representing a sequence of code pieces to be executed at certain points in time.

**Definition 2.2.1 (Job Sequence).** A sequence \( \sigma = [(R_0, e_0, v_0), (R_1, e_1, v_1), \ldots] \) is a job sequence if all jobs are monotonically ordered by release times, i.e., \( R_i \leq R_j \) for \( i \leq j \). Job sequences may be finite or infinite.

We further define absolute release time \( R(\sigma) \), accumulated execution time \( e(\sigma) \) and absolute deadline \( D(\sigma) \) of a job sequence as

\[
R(\sigma) := R_0, \quad e(\sigma) := \sum_{i \geq 0} e_i, \quad D(\sigma) := \sup_{i \geq 0} (R_i + d(v_i)).
\]

Intuitively, a job sequence is generated by a task \( T \) if it is consistent with some path in the graph \( G(T) \) in the following sense.

**Definition 2.2.2 (Generated Job Sequence).** For a task \( T \), a job sequence \( \sigma = [(R_0, e_0, \pi_0), (R_1, e_1, \pi_1), \ldots] \) is generated by \( T \), if \( \pi = (\pi_0, \pi_1, \ldots) \) is a path in \( G(T) \) and for all \( i \geq 0 \):

1. \( R_{i+1} - R_i \geq p(\pi_i, \pi_{i+1}) \) and
2. \( e_i \leq e(\pi_i) \).

We denote the set of all job sequences generated by \( T \) with \( \llbracket T \rrbracket \).

This definition allows finite as well as infinite executions. Further, note that the difference in release times of two successive jobs may be larger than the edge labels specify. This *sporadic* behavior of the model is intended in order to subsume other models with sporadic semantics. However, for all analysis methods presented in this thesis, it is safe to assume that jobs are released as early as possible, since that represents the worst-case behavior.
Example 2.2.3. For the example task $T$ in Figure 2.2, consider the job sequence $\sigma = [(5,5,v_4),(25,0.5,v_2),(42,3,v_3)]$. Clearly, $\pi = (v_4,v_2,v_3)$ is a path in $G(T)$ and release times and execution times satisfy the vertex and edge labels. Thus, $\sigma$ is generated by $T$.

Note that this example demonstrates the sporadic behavior allowed by the semantics of our model. While the second job in $\sigma$ (associated with $v_2$) is released as early as possible after the first job ($v_4$), the same is not true for the third job ($v_3$). Further, the first and third job take their maximum execution time while the second job does not.

The example illustrates that $T$ does not have an explicit start vertex. Job sequences may correspond to paths starting anywhere in the graph. □

Finally, a job sequence is generated by a DRT task system if it is a composition of job sequences generated by all its tasks, as follows. For a sequence $S$ and a set $A$, we use $S|_A$ to denote the projection of $S$ to elements of $A$, i.e., we retain only those components of $S$ that are included in $A$.

Definition 2.2.4 (Composed Job Sequence). For a task set $\tau = \{T_1, \ldots, T_N\}$, a job sequence $\sigma = [(R_0,e_0,\pi_0),(R_1,e_1,\pi_1),\ldots]$ is a composition of job sequences $\{\sigma_T\}_{T \in \tau}$ if $\sigma_{[R \times R \times V(T)]} = \sigma_T$ for all $T \in \tau$.

If $\sigma_T$ is generated by $T$ for all $T \in \tau$, we say that $\sigma$ is generated by $\tau$. We denote the set of all job sequences generated by $\tau$ with $[\tau]$. □

This definition implies an important property of DRT task systems which is that tasks behave independently from each other. This enables analysis procedures which are compositional.

Note that our definitions assume dense time semantics, i.e., job releases and durations are real numbers. However, since all vertex and edge labels are natural numbers, we will see that we usually only need to consider integers in all analysis methods focusing on worst-case behavior. Cases where non-integral numbers matter are explicitly stated.

2.3 Schedulers and System Model

When several jobs are ready to be executed at the same point in time, a scheduler needs to decide which job to run first. We assume a preemptive scheduler, i.e., it may suspend a job that has already been running for some time in favor of running another job first. We distinguish between dynamic and static priority schedulers, i.e., whether the scheduler has to obey a certain order of relative priorities on the task set. Generally, dynamic priority schedulers have more freedom in their scheduling decisions than static priority schedulers and can therefore successfully schedule more task sets.
In particular, we focus our analysis on two scheduling algorithms, earliest deadline first (EDF) schedulers and static priority (SP) schedulers.

**EDF scheduler.** The scheduler picks the job with the smallest absolute deadline, ties broken arbitrarily. This dynamic priority scheduling strategy is shown to be optimal for our setting of independent jobs, i.e., if a task set can be scheduled with any scheduler, it can also be scheduled with EDF.

**SP scheduler.** Given a priority order $Pr : \tau \rightarrow \mathbb{N}$ which assigns a unique priority to each task, the scheduler picks the job for execution that was released by the task $T$ with highest priority, i.e., minimal $Pr(T)$.

Note that the job model as the fundamental entity representing computation is so abstract that our results are not restricted to be only applied to computational resources. The results can be applied directly to other settings, as long as they can be expressed with the proposed job and task models and assume a dedicated uniprocessor resource.

### 2.4 Schedulability and Feasibility

Given the description of a system as a task set, we may want to know whether it is possible to schedule all behaviors of the system such that all timing constraints will always be met. In particular, given a scheduler, will all jobs always meet their deadlines? More generally: is there any scheduler capable of doing so? These concepts are called schedulability and feasibility.

**Definition 2.4.1 (Schedulability).** A task set $\tau$ is schedulable with scheduler $Sch$ (or $Sch$ schedulable) if and only if for all job sequences generated by $\tau$, all jobs meet their deadlines when scheduled with $Sch$. Otherwise, $\tau$ is unschedulable with $Sch$. □

**Definition 2.4.2 (Feasibility).** A task set $\tau$ is feasible if and only if there is a scheduler $Sch$ such that $\tau$ is schedulable with $Sch$. □

We may also restrict feasibility to SP schedulers when a priority order is not fixed, as follows.

**Definition 2.4.3 (SP Feasibility).** A task set $\tau$ is SP feasible if and only if there is a priority order $Pr : \tau \rightarrow \mathbb{N}$ such that $\tau$ is SP schedulable with that priority order $Pr$. □

Algorithms for testing feasibility or schedulability may obtain results faster if they are over-approximate, i.e., allow one-sided error. In particular, a schedulability test (and analogously, a feasibility test) can be:
Sufficient if it is failed by all non-schedulable task sets,
Necessary if it is satisfied by all schedulable task sets, or
Precise or exact if it is both sufficient and necessary.

In other words, if a task set satisfies a sufficient test, it is schedulable and if it fails a necessary test, it is non-schedulable. All results presented in this thesis focus on exact tests.
3. Feasibility Analysis

We present in this chapter a feasibility analysis method for DRT task systems with pseudo-polynomial run-time complexity. It is based on an iterative graph exploration technique that uses path abstractions in order to achieve the complexity result. Further, a few crucial optimizations are discussed which make this method very efficient in practice. The example task from Figure 2.2 on page 17 is used as a running example throughout the chapter.

While feasibility is not favoring a particular type of scheduler, the EDF scheduler plays a special role. It marks an interesting connection between schedulability and feasibility: EDF schedulability is equivalent to feasibility in our setting of independent jobs. More precisely, this holds for all task systems generating job sequences such that each job, once it is released, can be scheduled and run to completion. We establish this property in the next section together with a well-known characterization of feasibility using demand-bound functions. This characterization is the basis of our feasibility tests in the rest of this chapter.

3.1 The Feasibility Theorem

In order to prepare for the theorem and its proof, we need to define an abstraction of DRT tasks. Suitable abstractions play a key role in most of the results in this thesis. They are not only technical tools, but also deliver insights about the essential information necessary to investigate particular properties.

3.1.1 Demand-Bound Functions

A demand-bound function [6] is used to express the accumulated execution time that a task set can demand from the processor within any time interval of given length. In particular, it considers each execution requirement that is both released within the interval and needs to be finished before the end of the interval.

**Definition 3.1.1 (Demand-Bound Function).** For a task $T$ or a task set $\tau$ and an interval length $t$, $dbf_T(t)$ and $dbf_\tau(t)$ denote the maximal cumulative execution requirement of jobs with both release time and deadline within any interval of length $t$, over all job sequences generated by $T$ or $\tau$, respectively:

$$dbf_T(t) := \max \{ e(\sigma) \mid \sigma \in [T] \text{ and } D(\sigma) - R(\sigma) \leq t \},$$

$$dbf_\tau(t) := \max \{ e(\sigma) \mid \sigma \in [\tau] \text{ and } D(\sigma) - R(\sigma) \leq t \}.$$
Proposition 3.1.2. From the definition it is clear that \( dbf \) is additive in tasks:

\[
\text{dbf}_\tau(t) = \sum_{T \in \tau} \text{dbf}_T(t).
\]

This property relies on the independence of tasks from each other. Further, \( \text{dbf}_\tau \) is tight in the sense that for each \( t \), there is a job sequence generated by task set \( \tau \) in which some jobs actually have an execution demand of \( \text{dbf}_\tau(t) \) within an interval of \( t \) time units. Note that \( \text{dbf}_\tau \) and \( \text{dbf}_T \) are defined for all \( t \in \mathbb{R}_{\geq 0} \) since we assume time to be dense. However, changes clearly only occur at integer points, making the demand-bound functions that we consider in this thesis right-continuous step-functions.

Example 3.1.3. Consider job sequence \( \sigma = [(5, v_4), (25, 1, v_2), (42, 3, v_3)] \) generated by task \( T \) from Figure 2.2 on page 17. This sequence \( \sigma \) shows that in a time interval of size \( t = 45 \), task \( T \) may generate a demand of 9 on the processor as follows: The first job is released at \( R_1 = 5 \) and the third job has its deadline at \( R_3 + d(v_3) = 42 + 8 = 50 \). Thus, all three jobs of the sequence have both their release time and deadline within time interval \([5, 50]\) of length 45. Together, their execution time is \( 5 + 1 + 3 = 9 \).

In fact, there are no job sequences generated by \( T \) with a higher demand within an interval of length 45. We can conclude that \( \text{dbf}_T(45) = 9 \) for our example task \( T \). The demand-bound function of \( T \) on the interval \([0, 65]\) is plotted in Figure 3.1.

\[0, 10, 20, 30, 40, 50, 60\quad t\]
\[0, 3, 6, 9, 12\quad \text{dbf}_T(t)\]

\[\text{Figure 3.1. Demand-bound function for the DRT task in Figure 2.2.}\]

3.1.2 Feasibility Characterization

We can use the demand-bound function to show equivalence of EDF schedulability and feasibility with the following theorem. The equivalent characterization of feasibility based on the demand-bound function is used in our feasibility analysis algorithm in the following sections.
Theorem 3.1.4 (Feasibility Theorem). For a task set $\tau$, the following three properties are equivalent.

1. Task set $\tau$ is feasible.
2. Task set $\tau$ is EDF schedulable.
3. The following condition holds: $\forall t \geq 0 : \text{dbf}_\tau(t) \leq t$

Proof. We establish the proof by showing $\neg 1. \implies \neg 2. \implies \neg 3. \implies \neg 1.$ The proof is similar to the one presented in [12].

$\neg 1. \implies \neg 2.$: If $\tau$ is not feasible, it can obviously not be scheduled by EDF.

$\neg 2. \implies \neg 3.$: Assume that $\tau$ is not schedulable with EDF. There must be a job sequence $\sigma \in [\tau]$ in which a job $J$ misses its deadline if $\sigma$ is scheduled with EDF. Let $\sigma$ be minimal, i.e., if any job is removed from $\sigma$, then all jobs in $\sigma$ meet their deadlines under an EDF scheduler. From this, we derive two properties.

First, consider the deadline of job $J$. EDF assigns priorities to jobs after absolute deadlines, so the deadline of $J$ must be equal to $D(\sigma)$, the latest absolute deadline of any job in $\sigma$. Otherwise, the job in $\sigma$ with deadline $D(\sigma)$ would have a lower priority than $J$ and could therefore be removed from $\sigma$ without removing the deadline miss of $J$.

Second, the processor must be busy at all time points in the interval $[R(\sigma), D(\sigma)]$, since otherwise, all jobs $J'$ for which there is an idle instant between the releases of $J'$ and $J$ could be removed from $\sigma$ without removing the deadline miss of $J$.

Taking both properties together with the assumption that $J$ misses its deadline at $D(\sigma)$, we see that all jobs in $\sigma$ are released and have their deadline in the interval $[R(\sigma), D(\sigma)]$. Their execution requirement exceeds the length of the interval, $D(\sigma) - R(\sigma)$, since the processor is occupied throughout the whole interval, and $J$ is not done at $D(\sigma)$. Therefore, $\text{dbf}_\tau(D(\sigma) - R(\sigma)) > D(\sigma) - R(\sigma)$, contradicting property 3.

$\neg 3. \implies \neg 1.$: Assume that property 3 of Theorem 3.1.4 is not satisfied for a task set $\tau$, i.e., there is $t \geq 0$ such that $\text{dbf}_\tau(t) > t$. We show that $\tau$ cannot be feasible by noting that because of the tightness of $\text{dbf}_\tau$, there is a time interval of size $t$ during which $\tau$ may release a job sequence $\sigma$ with more than $t$ time units of accumulated execution demand. Further, all jobs in $\sigma$ have their deadlines inside the interval. Thus, more than $t$ time units of execution demand need to be executed within $t$ time units. That is clearly impossible for any scheduler, making $\tau$ infeasible.

This theorem means that we can move freely between feasibility and EDF schedulability. The feasibility test developed in this chapter is therefore automatically an EDF schedulability test.
Further, note that the assumptions for this theorem are very general. Jobs are assumed to be able to start executing and run to completion as soon as they are released, being preemptable at any time. A DRT task system satisfies these assumptions, but they are more general, so that the Feasibility Theorem can be applied to extensions of the DRT model later in this thesis.

3.2 An Iterative Solution

The characterization of feasibility in Theorem 3.1.4 using the demand-bound function can be used as a basis for a practical feasibility test.

\[ \forall t \geq 0 : dbf_\tau(t) \leq t \]  

(3.1)

Feasibility can be checked by determining whether there exists a \( t_f \) violating this property, i.e., such that \( dbf_\tau(t_f) > t_f \). In order to check this, we need to solve two main problems:

1. How do we compute \( dbf_\tau(t) \) for a given \( t \)?
2. For which time interval sizes \( t \) do we need to compute \( dbf_\tau(t) \)?

We develop an iterative method of computing \( dbf_\tau(t) \) for given \( t \) in Sections 3.2.1 and 3.2.2. The second problem is important as well: if we were to naïvely check \( dbf_\tau(t) \) sequentially for all integers \( t \), we would not terminate when analyzing feasible task systems. We solve this in Section 3.2.3 by deriving a pseudo-polynomial bound for \( t_f \).

A technical assumption in this section is that all deadlines are monotonic absolute. An extension of the method to arbitrary deadlines is presented in Section 3.3.

3.2.1 Demand Pairs as Path Abstractions

Using Proposition 3.1.2, the first problem of how to compute \( dbf_\tau(t) \) reduces to calculating \( dbf_T(t) \) for each task \( T \). Recall Definition 3.1.1 of the demand-bound function. It takes the maximum over all job sequences \( \sigma \in [T] \) with a difference \( D(\sigma) - R(\sigma) \leq t \) between release time of the first job and latest deadline of any job. Since every job sequence \( \sigma \in [T] \) is generated by a path \( \pi \) through \( G(T) \), we can express the demand-bound function in terms of all paths \( \pi \). For doing so, we extend the notions of execution time demand and relative deadline from single jobs to job sequences as follows.

**Definition 3.2.1 (Demand Pair).** For a finite path \( \pi = (\pi_0, \ldots, \pi_l) \) in \( G(T) \) we define:
Execution demand: \( e(\pi) := \sum_{i=0}^{l} e(\pi_i) \)

Deadline: \( d(\pi) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + d(\pi_l) \)

We call \((e(\pi), d(\pi))\) a demand pair that corresponds to \(\pi\).

Using this notation, we can express \(\text{dbf}_T\) directly in terms of paths:

\[
\text{dbf}_T(t) = \max \{ e(\pi) \mid \pi \in \text{Paths}(G(T)) \text{ and } d(\pi) \leq t \}.
\]

Note that this characterization implies that for each \(t\), the value \(\text{dbf}_T(t)\) is equal to the execution demand \(e(\pi)\) of some path \(\pi\) in \(G(T)\). We say that \(\pi\) certifies or witnesses \(\text{dbf}_T(t)\).

A naïve approach for computing \(\text{dbf}_T(t)\) could be to enumerate all paths \(\pi\) up to deadline \(d(\pi) \leq t\). However, this would be an exponential procedure. A key idea for efficiency is to use demand pairs as path abstractions and explore the space of demand pairs instead of enumerating all paths. In particular, the demand-bound function can be expressed directly in terms of demand pairs:

\[
\text{dbf}_T(t) = \max \{ e \mid (e, d) \text{ demand pair with } d \leq t \}.
\]

If we were able to compute all demand pairs, we could easily compute the demand-bound function.

**Example 3.2.2.** In Figure 3.2, \(\text{dbf}_T\) for the task in Figure 2.2 on page 17 is again (partially) shown. Note that it only changes at certain points, since \(\text{dbf}_T\) is a step function. Each point corresponds to a path \(\pi\) in \(G(T)\). Take for example the marked point \(t = 43\) with \(\text{dbf}_T(43) = 9\). It corresponds to path \(\pi = (v_4, v_2, v_3)\) from Example 3.1.3. As we have seen there, \(\pi\) has an execution requirement of 9 time units. Further, if all jobs are released as early as possible, the time interval between the release of the first job and the deadline of the third job is 43 time units. Therefore, \(\pi\) is abstracted by demand pair \(\langle 9, 43 \rangle\).

### 3.2.2 Iterative Procedure using Demand Triples

A key observation for calculating demand pairs efficiently is that several different paths may correspond to the same demand pair.
Figure 3.2. Demand-bound function for the DRT task in Figure 2.2. The diamonds depict the demand pairs of that task. Demand pairs corresponding to multiple paths have a darker color.

**Example 3.2.3.** In our running example from Figure 2.2 on page 17, consider the following three paths:

\[ \pi^A = (v_3, v_1, v_5, v_4) \]
\[ \pi^B = (v_3, v_1, v_2, v_4) \]
\[ \pi^C = (v_4, v_2, v_3, v_1) \]

All three paths correspond to the demand pair \( \langle 11, 51 \rangle \). □

Using demand pairs as an abstraction of concrete paths through the task graph reduces the number of objects that need to be tracked. In order to develop this into an efficient iterative procedure, we need to extend the abstraction. Note that in the above example, paths \( \pi^A \) and \( \pi^B \) (in contrast to \( \pi^C \)) not only correspond to the same demand pair, but also end in the same vertex of \( G(T) \). Any extension of both paths will therefore always result in the same demand pairs for both extended paths. In an iterative procedure, only one of them (or rather, an abstraction of both) needs to be kept. This is the key to preventing an exponential explosion in complexity.

We formalize these observations as follows.

**Definition 3.2.4 (Demand Triple).** For a finite path \( \pi = (\pi_0, \ldots, \pi_l) \) in \( G(T) \), we call \( \langle e(\pi), d(\pi), \pi_l \rangle \) a demand triple. We say that \( \pi \) corresponds to that demand triple. □

**Lemma 3.2.5.** For each constant \( D \in \mathbb{N} \), the number of demand triples \( \langle e, d, v \rangle \) with \( d \leq D \) is bounded polynomially in \( D \) and \( n \), the number of vertices in \( G(T) \). □

**Proof.** We may assume \( e \leq d \) since otherwise there is clearly a deadline violation. Thus, all demand triples are in \( \mathbb{N}_{\leq D} \times \mathbb{N}_{\leq D} \times \mathbb{N}_{\leq n} \), leaving only \( O(D^2 n) \) possibilities. □
Using these insights, we can implement an iterative procedure for calculating $\text{dbf}_T(t)$ for a given $t$ using a graph traversal that stores only demand triples. We first give a high-level description:

1. Consider all paths of length 0, i.e., all vertices of $G(T)$, and store their corresponding demand triples.

2. For a stored demand triple $\langle e, d, u \rangle$, consider all successor vertices $v$ of $u$. For each such $v$, compute a new demand triple $\langle e', d', v \rangle$ corresponding to a path that has been extended by $v$. In particular,
   $$e' = e + e(v) \quad \text{and} \quad d' = d - d(u) + p(u,v) + d(v).$$

   If $\pi$ is a path corresponding to $\langle e, d, u \rangle$, then this computation ensures that $\langle e', d', v \rangle$ corresponds to $\pi$ extended by vertex $v$. See Figure 3.3.

3. Each newly computed demand triple $\langle e', d', v \rangle$ is stored only if:
   - It is not stored yet, and
   - $d' \leq t$.

4. Repeat until there are no new demand triples.

5. Using all demand triples $\langle e, d, v \rangle$, calculate $\text{dbf}_T(t)$:
   $$\text{dbf}_T(t) = \max \{ e \mid \langle e, d, v \rangle \text{ demand triple with } d \leq t \}.$$ 

Note that the above procedure only needs to discover paths $\pi$ (or rather, the demand triple abstractions thereof) which have a path length of at most $t$, since $d(\pi) \geq k$ for every $k$-path because of non-zero edge labels. We give a formal, more detailed algorithm in Figure 3.4 which is based on this observation. For simplicity of presentation, it discovers all paths in a breadth-first manner. In particular, the sets $DT_k$ contain all demand triples corresponding to paths of length $k$. 

![Figure 3.3](image-url) Extending a demand triple $\langle e, d, u \rangle$ by a vertex $v$. Let $\langle e, d, u \rangle$ correspond to a path $\pi = (\pi_0, \ldots, \pi_{l-1}, u)$ and $\langle e', d', v \rangle$ correspond to its extension $\pi' = (\pi_0, \ldots, \pi_{l-1}, u, v)$. Depicted is a job release sequence generated by $\pi'$ in which all jobs are released as early as possible. The relation between $d$ and $d'$ is shown.
function compute-dbft(t):
1: \( DT_0 \leftarrow \{ \langle e(v), d(v), v \rangle \mid v \in V(T) \} \)
2: for \( k = 1 \) to \( t \) do
3: \( DT_k \leftarrow \emptyset \)
4: for all \( \langle e, d, u \rangle \in DT_{k-1} \) do
5: for all edges \((u, v)\) in \( G(T) \) do
6: \( e' \leftarrow e + e(v) \)
7: \( d' \leftarrow d - d(u) + p(u, v) + d(v) \)
8: if \( d' \leq t \) then
9: \( DT_k \leftarrow DT_k \cup \{ \langle e', d', v \rangle \} \)
10: end if
11: end for
12: end for
13: end for
14: return \( \max \{ e \mid \langle e, d, v \rangle \in \bigcup_{k \leq t} DT_k \} \)

Figure 3.4. An iterative algorithm for computing \( dbf_T(t) \).

Correctness

By the intuitive reasoning above, the algorithm in Figure 3.4 correctly tracks all paths through \( G(T) \). It only discards paths if their deadline already is too long, or if it discovers a path corresponding to an already considered demand triple. We formalize these insights as follows.

**Lemma 3.2.6.** The algorithm in Figure 3.4 is correct. In particular:

1. There is a path \( \pi \) in \( G(T) \) with \( d(\pi) \leq t \) for which \( e(\pi) \) is equal to the return value.
2. There is no path \( \pi' \) in \( G(T) \) with \( d(\pi') \leq t \) for which \( e(\pi') \) strictly exceeds the return value.

**Proof.** For the first part, let \( e \) be the output of our algorithm in Figure 3.4 given input \( t \). We want to show that there is a path \( \pi \) in \( G(T) \) with \( e(\pi) = e \) and \( d(\pi) \leq t \). From the return statement (line 14) it follows that there must have been a demand triple \( \langle e, d, v \rangle \in DT_k \) for some \( k \leq t \) with \( d \leq t \). It is thus sufficient to show that for each demand triple in each \( DT_k \), there is a corresponding path of length \( k \) in \( G(T) \). This can be shown by induction on \( k \):

- \( k = 0 \): All demand triples in \( DT_0 \) are generated in line 1 directly from the vertices of \( G(T) \) which are 0-paths.
- \( k - 1 \leadsto k \): Given \( \langle e', d', v \rangle \in DT_k \), it must have been added to \( DT_k \) in line 9. The demand triple \( \langle e, d, u \rangle \) from which it was created must have been chosen in line 4 from \( DT_{k-1} \). By induction hypothesis, there is a path
\(\pi = (\pi_0, \ldots, \pi_{k-2}, u)\) of length \(k - 1\) in \(G(T)\) corresponding to \(\langle e, d, u \rangle\). Further, there is an edge \((u, v)\) in \(G(T)\) that has been chosen in line 5. Thus, \(\pi' = (\pi_0, \ldots, \pi_{k-2}, u, v)\) is a path of length \(k\) in \(G(T)\). Finally, the calculations in lines 6 and 7 ensure that \(\pi'\) corresponds to \(\langle e', d', v \rangle\) because of monotonic absolute deadlines.

For the second part, it is sufficient to show that all paths \(\pi\) of length \(k\) in \(G(T)\) with a deadline of at most \(t\) have a corresponding demand triple \(\langle e, d, v \rangle\) in \(DT_k\). Again, we show this by induction on \(k\).

\(k = 0\): For all 0-paths, i.e., all vertices, all demand triples are included in \(DT_0\) via line 1.

\(k - 1 \rightsquigarrow k\): Given a path \(\pi = (\pi_0, \ldots, \pi_{k-2}, u, v)\) of length \(k\) that has a deadline \(d(\pi) \leq t\), we know by induction hypothesis that \(\pi' = (\pi_0, \ldots, \pi_{k-2}, u)\) of length \(k - 1\) has a corresponding demand triple \(\langle e, d, u \rangle\) in \(DT_{k-1}\). This demand triple is eventually chosen in line 4 and also the edge \((u, v)\) in line 5. Finally, lines 6 and 7 calculate the corresponding demand triple for \(\pi\) which will be added to \(DT_k\) in line 9 since \(d' = d(\pi) \leq t\) by assumption.

**Complexity**

The demand triple abstraction ensures that for each \(k\), there can be at most polynomially many demand triples in \(DT_k\) (Lemma 3.2.5). This bounds the number of iterations of line 4. Further, for each demand triple, there are at most \(n\) extensions (maximal number of successors), which also bounds number of iterations of line 5. Finally, finding the maximum in line 14 can be done in time linear in the set size. We summarize the result in the following lemma.

**Lemma 3.2.7.** The run-time of the algorithm in Figure 3.4 is bounded polynomially in \(t\) and \(n\).

### 3.2.3 Calculating the Bound

We will now derive a bound for \(t_f\), the supposed counterexample for Condition (3.1) for infeasible task sets.

With such a bound \(D\), we could run the following feasibility test. Given a task set \(\tau\), check whether \(dbf_\tau(t) \leq t\) for all \(t \leq D\). If and only if the test succeeds for all \(t \leq D\), then \(\tau\) is feasible. Otherwise, a counterexample \(t_f\) is found and \(\tau\) is shown to be infeasible. Further, if we can show that \(D\) is polynomially bounded in the parameters of the task set, the sketched test is polynomially bounded as well.
Utilization

A central concept we use is the utilization of a task set. Intuitively, it describes the maximal execution demand rate that a task set may create asymptotically.

Definition 3.2.8 (Utilization). For a task $T$ and a task set $\tau$, we define their utilizations:

$$U(T) := \lim_{t \to \infty} \frac{dbf_T(t)}{t}, \quad U(\tau) := \lim_{t \to \infty} \frac{dbf_{\tau}(t)}{t}.$$ 

Further, for a cycle $\pi = (\pi_0, \ldots, \pi_l)$ in $G(T)$, i.e., $\pi_0 = \pi_l$ and $l \geq 1$, we define:

$$U(\pi) := \frac{\sum_{j=0}^{l-1} e(\pi_j)}{\sum_{j=0}^{l-1} p(\pi_j, \pi_{j+1}).}$$

The limits in the above definition exist because the demand-bound function is a monotonically increasing step function for which the steps have bounded “heights” and occur at integer points. Therefore, the utilization is well-defined.

Since the demand-bound function is additive, the utilization also is, i.e., $U(\tau) = \sum_{T \in \tau} U(T)$. Further, the utilization of tasks is related to the utilization of the cycles in their graphs, as follows. We will see below in Section 3.2.4 how this can be used to practically compute the utilization.

Lemma 3.2.9 (Cycle Characterization). For a task set $\tau$ and a task $T$, we have

$$U(T) = \max \{ U(\pi) \mid \pi \text{ is a cycle in } G(T) \}, \quad (3.2)$$

$$U(\tau) = \sum_{T \in T} U(T). \quad (3.3)$$

We call the cycle with utilization $U(T)$ the most dense cycle in $G(T)$. \hfill \Box

Proof. Condition (3.3) follows directly from the additivity of $dbf$. We prove Condition (3.2) in the following two steps if $G(T)$ contains a cycle. (Otherwise, the result is trivial with $\max \emptyset := 0.$)

$$\liminf_{t \to \infty} \frac{dbf_T(t)}{t} \geq \max \{ U(\pi) \mid \pi \text{ is a cycle in } G(T) \}, \quad (3.4)$$

$$\limsup_{t \to \infty} \frac{dbf_T(t)}{t} \leq \max \{ U(\pi) \mid \pi \text{ is a cycle in } G(T) \}. \quad (3.5)$$

Proof of (3.4). Intuitively, we can take a cycle of maximal utilization and repeat it over and over, creating a lower bound for $dbf_T$ and thus leading to a lower bound for any limit point of $dbf_T(t)/t$.

Formally, let $\pi$ be a cycle in $G(T)$ with maximal $U(\pi)$. We define a sequence $\{(e_i, d_i)\}_{i \in \mathbb{N}}$ of demand pairs with

$$\langle e_i, d_i \rangle := \langle e(\pi^i), d(\pi^i) \rangle,$$
where $\pi'$ is $\pi$ repeated for $i$ times. Clearly, all $\langle e_i, d_i \rangle$ are demand pairs for $T$ and therefore, $\text{dbf}_T(d_i) \geq e_i$. Further, $\text{dbf}_T$ is monotonically increasing, so $\text{dbf}_T(t) \geq e_i$ for all $t \in [d_i, d_i + p(\pi)]$ since $p(\pi) = d_{i+1} - d_i$. Together, it follows that

$$\liminf_{t \to \infty} \frac{\text{dbf}_T(t)}{t} \geq \liminf_{i \to \infty} \frac{\text{dbf}_T(d_i)}{d_i + p(\pi)} \geq \lim_{i \to \infty} \frac{e_i}{d_i + p(\pi)} = U(\pi).$$

**Proof of (3.5).** Proof by contradiction. Intuitively, if $\text{dbf}_T(t)/t$ has a limit point strictly larger than the maximal utilization of any cycle, then for a sufficiently large $t$, there is a path $\pi$ certifying the value of $\text{dbf}_T(t)$ with the following property. The path $\pi$ must contain cycles of which at least one has a utilization arbitrarily close to the largest limit point of $\text{dbf}_T(t)/t$. This, however, contradicts the initial assumption.

Formally, let $\gamma$ be a limit point of $\text{dbf}_T(t)/t$ for $t \to \infty$ and we assume

$$\gamma > \max \{ U(\pi) \mid \pi \text{ is a cycle in } G(T) \}.$$

In particular, for every $\varepsilon > 0$, there must be an infinite and monotonically increasing sequence of natural numbers $\{t_i\}_{i \in \mathbb{N}}$ such that $\text{dbf}_T(t_i)/t_i \geq \gamma - \varepsilon$. Let $e^{\text{sum}}(T) := \sum_{v \in V(T)} e(v)$ and $d^{\text{min}}(T) := \min_{v \in V(T)} d(v)$. Since $t_i$ and $\text{dbf}_T(t_i)$ tend to infinity, there must be $t_i$ such that:

$$\frac{\text{dbf}_T(t_i) - e^{\text{sum}}}{t_i - d^{\text{min}}} \geq \gamma - 2 \cdot \varepsilon.$$

Let $\pi$ be a path certifying $\text{dbf}_T(t_i)$, i.e., $e(\pi) = \text{dbf}_T(t_i)$ and $d(\pi) \leq t_i$. This is the path from our intuitive introduction of the proof from which we will extract a cycle with high utilization as follows. Let $\pi^*$ be what remains of $\pi$ if we cut out all cycles. (If $\pi$ does not contain any cycles, choose a larger $t_i$.) Since $\pi^*$ is simple, we have for the rest of $\pi$:

$$\frac{e(\pi) - e(\pi^*)}{d(\pi) - d(\pi^*)} \geq \frac{\text{dbf}_T(t_i) - e^{\text{sum}}}{t_i - d^{\text{min}}} \geq \gamma - 2 \cdot \varepsilon.$$

Note that $e(\pi) - e(\pi^*)$ is the sum of all vertices in all cycles in $\pi$, and $d(\pi) - d(\pi^*)$ is the accumulated duration of all cycles in $\pi$, i.e., the sum of their edge labels. For sufficiently small $\varepsilon$, their quotient is strictly larger than $\max \{ U(\pi) \mid \pi \text{ is a cycle in } G(T) \}$ by the initial assumption for $\gamma$. However, this implies that at least one cycle in $\pi$ must have a utilization that is strictly larger as well, leading to a contradiction.

The last step holds since for all $a_i, b_i, c, d \in \mathbb{N}_{>0}$, a simple induction over $i \in \mathbb{N}$ shows that

$$\forall i : \frac{a_i}{b_i} \leq \frac{c}{d} \implies \sum_i a_i \leq \frac{c}{\sum_i b_i} \leq \frac{c}{d}.$$
The cycle characterization of a task set’s utilization allows us to work with the rather concrete concept of cycles through graphs instead of the more abstract notion of the limit as in the original definition.

Clearly, if the utilization of a task set exceeds 1, then it cannot be schedulable, since after sufficiently long cycling of each task in its most dense cycle, the system will be overloaded. On the other hand, if the utilization is smaller than 1, we can show that for sufficiently long intervals, the execution demand will be strictly less than the interval size\(^1\). For an intuitive explanation, see Figure 3.5.

![Figure 3.5](image)

*Figure 3.5. Demand-bound function of some task set. This task set is not schedulable since there is \(t_f\) with \(dbf(t_f) > t_f\). Let’s assume there is a linear bound for \(dbf(t)\) with a slope of less than 1. This bound intersects \(t\) at some point \(D\). Clearly, \(t_f\) is at most \(D\), so only values up to \(D\) need to be checked. For the slope of the bound, we use the task set’s utilization.*

Formally, we first derive a bound for the demand-bound function of each individual task. Each path in \(G(T)\) can be partitioned into two types of sub-paths: cycles of maximal length and all remaining vertices. For all maximal cycles, their execution demand can be bounded by the demand that the most dense cycle could generate in the same amount of time. Further, each vertex of \(G(T)\) belongs at most once to the remaining vertices (otherwise, some cycle was not chosen maximal), therefore a bound for the execution demand of the remaining vertices is just the sum of all execution times of all vertices\(^2\).

We express this observation in the following lemma, using \(e^{\text{sum}}(T) := \sum_{v \in V(T)} e(v)\).

**Lemma 3.2.10 (Linear \(dbf\) Bound).** For a task \(T\) and all \(t \in \mathbb{R}_{\geq 0}\),

\[
dbf_T(t) \leq t \cdot U(T) + e^{\text{sum}}(T).\]

\(\square\)

\(^1\) Note that this does not mean that \(U(\tau) < 1\) is sufficient for feasibility. It only guarantees the existence of a bound \(D\) for \(t_f\).

\(^2\) These bounds can of course be tightened with the potential of improving analysis performance. However, for simplicity of presentation, we use the presented bound, which is already sufficient for the complexity result.
Proof. For a given \( t \), let \( \pi \) be a path with maximal execution demand during any interval of length \( t \), i.e., \( e(\pi) = dbf_T(t) \) and \( d(\pi) \leq t \). If \( \pi \) is simple, we have \( e(\pi) \leq e^{\text{sum}}(T) \) and are done. Otherwise, we first identify all cycles in \( \pi \) as follows. For each vertex \( v \) in \( G(T) \), let \( m(v) \) denote the number of occurrences of \( v \) in \( \pi \). Find the first vertex \( \pi_i \) in \( \pi \) with \( m(\pi_i) > 1 \). Let \( \pi_j \) be the last occurrence of \( \pi_i \) in \( \pi \). We have \( i < j \) and call \( \pi^{(1)} = (\pi_i, \ldots, \pi_j) \) the first cycle in \( \pi \). We remove it from \( \pi \) by considering the new path 

\[
\pi' = (\pi_0, \ldots, \pi_i, \pi_{i+1}, \ldots, \pi_l).
\]

Note that we cut away the edges of \( \pi^{(1)} \) from \( \pi \) which only leaves the start vertex \( \pi_i \) (that equals \( \pi_j \)) of \( \pi^{(1)} \) in \( \pi \).

We repeat this procedure until no new cycles are found. Let \( \pi^{(j)} \) denote the \( j \)-th cycle found by this procedure and \( \pi^* \) the resulting path (\( \pi \) with all cycles removed). Clearly, \( \pi^* \) is simple. Note that the \( \pi^{(j)} \) do not need to be simple, i.e., they may contain sub-cycles.

For each \( \pi^{(j)} \), let \( l_j \) denote its length. By construction, we know now (via reordering) that

\[
e(\pi) = e(\pi^*) + \sum_j \sum_{i=0}^{l_j-1} e\left(\pi^{(j)}_i\right).
\]

Clearly, \( e(\pi^*) \leq e^{\text{sum}}(T) \) since \( \pi^* \) is simple. Further, for all cycles \( \pi^{(j)} \) we have

\[
\sum_{i=0}^{l_j-1} e\left(\pi^{(j)}_i\right) = U\left(\pi^{(j)}\right) \cdot \sum_{i=0}^{l_j-1} p\left(\pi^{(j)}_i, \pi^{(j)}_{i+1}\right)
\]

\[
\leq U(T) \cdot \sum_{i=0}^{l_j-1} p\left(\pi^{(j)}_i, \pi^{(j)}_{i+1}\right).
\]

This holds because \( U(T) \) is the maximal utilization over all cycles. Summing over all cycles of \( \pi \), we get

\[
\sum_j \sum_{i=0}^{l_j-1} e\left(\pi^{(j)}_i\right) \leq U(T) \cdot \sum_{j=0}^{l_j-1} \sum_{i=0}^{l_j-1} p\left(\pi^{(j)}_i, \pi^{(j)}_{i+1}\right)
\]

\[
\leq U(T) \cdot t.
\]

Finally, we apply this to Equation (3.6) and we get

\[
dbf_T(t) = e(\pi) \leq t \cdot U(T) + e^{\text{sum}}(T).
\]
The above bound for $dbf_T(t)$ for all tasks $T$ can be used to derive an upper bound for a witness $t_f$ with $dbf_{\tau}(t_f) > t_f$ as follows, similar to [11].

**Lemma 3.2.11 (Counterexample Bound).** For $\tau$ with $U(\tau) < 1$ which is not feasible, there is a $t_f$ with $dbf_{\tau}(t_f) > t_f$ such that:

$$t_f < \frac{\sum_{T \in \tau} e^{\text{sum}}(T)}{1 - U(\tau)} \quad \square$$

*Proof. Use Lemma 3.2.10 in $t_f < \sum_{T \in \tau} dbf_T(t_f)$ and simple arithmetics. ■*

Thus, only all $t$ up to the specified bound need to be checked. Further, for task sets with $U(\tau) \leq c$ for a constant $c < 1$, the bound guarantees that the feasibility problem is tractable by using our proposed algorithm. The main technical result of this chapter is stated in the following theorem.

**Theorem 3.2.12.** For a DRT task set $\tau$ with $U(\tau) \leq c$ for some constant $c < 1$, feasibility can be decided in pseudo-polynomial time. \quad \square

*Proof. We first note that for $U(\tau) \leq c$, we have

$$\sum_{T \in \tau} e^{\text{sum}}(T) \leq \frac{\sum_{T \in \tau} e^{\text{sum}}(T)}{1 - c}. $$

The term on the right-hand side is clearly polynomial in the values of the task parameters, so it is pseudo-polynomial in the system specification.

Further, given a $\tau$ with bounded utilization, we can compute $dbf_{\tau}(t)$ for all $t$ smaller than the bound for $t_f$ by running the algorithm in Figure 3.4 for each task $T$. Each time it is called, it returns $dbf_T(t)$ in time polynomial in $t$ and $n$ (Lemma 3.2.7). From the bound for $t_f$ it follows that the whole procedure is pseudo-polynomial. ■

Of course, efficient implementations can use an integrated procedure, which calculates all demand triples up to the given bound just once, while checking the condition $dbf_{\tau}(t) \leq t$. We discuss a few implementation and optimization details in Section 3.2.5.

### 3.2.4 Computing the Utilization

So far, we only used the utilization of tasks in definitions and proofs, but how can it actually be computed? In contrast to simpler task models, this question does not have a straightforward answer, since our notion of a task’s utilization is rather involved. We now present an efficient way of computing $U(T)$ based on an iterative procedure similar to the computation of $dbf_T(t)$ above in Section 3.2.2.
In general, we need to find the maximal utilization that a cycle $\pi$ in $G(T)$ can possibly have (Definition 3.2.8). The main problem is that there are potentially infinitely many cycles in $G(T)$, since the cycle characterization of $U(T)$ in Lemma 3.2.9 on page 32 is not restricted to simple cycles (that is, cycles which do not visit any vertex twice, except the last one). The key observation for an efficient construction is that it actually is sufficient to restrict the attention to simple cycles. The reason is that for any cycle, we can always find a simple one with a resulting utilization of no less than the original cycle, as follows. If the given cycle contains sub-cycles, then either at least one of them has a higher utilization, or all sub-cycles can be removed. We state this observation formally in the following lemma.

Lemma 3.2.13. For each cycle $\pi$, there is a simple cycle $\pi'$ with $U(\pi') \geq U(\pi)$.

Proof. Given a cycle $\pi = (\pi_0, \ldots, \pi_l)$, we show the existence of a simple cycle $\pi'$ with $U(\pi') \geq U(\pi)$ by induction on $l$. Clearly, a cycle of length $l = 1$ is simple, so the base case is trivial. For the induction step, we first identify the sub-cycles $\pi^{(j)}$ as in the proof of Lemma 3.2.10. (To be precise, this time $m(v)$ as used there does not consider the end vertex of $\pi$.) If there are no sub-cycles, we are done, since $\pi$ must be simple in that case. Otherwise, there are two cases for the utilizations of the sub-cycles:

First case: $\exists j : U\left(\pi^{(j)}\right) \geq U(\pi)$. In this case, since $\pi^{(j)}$ is shorter than $\pi$, we know by induction hypothesis that there is a simple cycle $\pi'$ with $U(\pi') \geq U\left(\pi^{(j)}\right)$ and we are done.

Second case: $\forall j : U\left(\pi^{(j)}\right) < U(\pi)$. We now know that all sub-cycles of $\pi$ have a lower utilization than $\pi$ itself. Thus, we construct $\pi'$ by removing all sub-cycles, i.e., $\pi'$ is the $\pi^*$ from the procedure of identifying the sub-cycles. For all $j$, let $l_j$ denote the length of $\pi^{(j)}$ as before, and $i_j$ denote the position of the first vertex of $\pi^{(j)}$ in $\pi$. With this we can write

$$\pi' = (\pi_0, \ldots, \pi_{i_j}, \pi_{i_j+1+1}, \ldots, \pi_l).$$

For its utilization, we get

$$U(\pi') = \frac{\sum_{i=0}^{l-1} e(\pi_i) - \sum_j \sum_{i=i_j}^{i_j+l_j-1} e(\pi_i)}{\sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) - \sum_j \sum_{i=i_j}^{i_j+l_j-1} p(\pi_i, \pi_{i+1})} \geq \frac{\sum_{i=0}^{l-1} e(\pi_i)}{\sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1})} = U(\pi).$$
Note that the last inequality holds since we have for all $a, b, c_i, d_i \in \mathbb{N}_{>0}$ with $a > \sum_i c_i > 0$ and $b > \sum_i d_i > 0$:
\[
\forall i : \frac{a}{b} \geq \frac{c_i}{d_i} \implies \frac{a - \sum_i c_i}{b - \sum_i d_i} \geq \frac{a}{b}
\]

**Example 3.2.14.** The digraph of the example task $T$ in Figure 2.2 on page 17 contains exactly the following simple cycles (up to rotation):

- $\pi^D = (v_1, v_2, v_3, v_1), \quad U(\pi^D) = 6/36$,
- $\pi^E = (v_1, v_5, v_4, v_2, v_3, v_1), \quad U(\pi^E) = 12/76,
- $\pi^F = (v_2, v_4, v_2), \quad U(\pi^F) = 6/40$.

According to Lemma 3.2.13 which we just showed, no other cycle in $G(T)$ can have a higher utilization. Consequently, we know that the utilization of $T$ is $U(T) = U(\pi^D) = 1/6$. □

The computation of $U(T)$ can be based on this observation, since we may restrict the search space for the worst-case cycle to cycles of at most length $n$. A naïve search via explicit enumeration would still be an exponential procedure. However, we can reuse our path abstraction framework which was already useful to reduce the complexity of the $\text{dbf}(t)$ computation. We need to adjust the demand triple abstraction idea so that:

1. $e$ now does not include the execution demand of the last vertex.
2. $d$ now does not include the deadline of the last vertex (but it does include the last inter-release separation). Therefore, we call it $p$ instead.
3. We include the start vertex, in addition to the end vertex.

Formally, we capture this as follows (see also Figure 3.6).

**Definition 3.2.15 (Utilization Triple).** For a finite path $\pi = (\pi_0, \ldots, \pi_l)$ in $G(T)$, we call $\langle e, p, (\pi_0, \pi_l) \rangle$ a utilization triple, if
\[
\begin{align*}
e &= \sum_{i=0}^{l-1} e(\pi_i) \quad \text{and} \quad p &= \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}).
\end{align*}
\]

**Example 3.2.16.** Consider path $\pi = (v_4, v_2, v_3)$ from the running example in Figure 2.2 on page 17. For calculating the utilization, it would be abstracted as the utilization triple $\langle 6, 35, (v_4, v_3) \rangle$. □

Using this abstraction, we are particularly interested in utilization triples $\langle e, p, (u, v) \rangle$ where start and end vertex coincide, $u = v$. In that case, they abstract a cycle and $e/p$ is the utilization of that cycle.
\[ \pi_0 \quad \pi_{l-1} \quad \pi_l \]

\[ d(\pi_0) \quad \ldots \quad d(\pi_{l-1}) \quad d(\pi_l) \]

\[ p \]

\[ t \]

**Figure 3.6.** Illustration of \( p \) in a utilization triple \( \langle e, p, (\pi_0, \pi_l) \rangle \) corresponding to path \( \pi = (\pi_0, \ldots, \pi_l) \). Note that if \( \pi_0 = \pi_l \), then \( \pi \) is a cycle and \( p \) is the duration of that cycle.

We summarize our algorithm for computing \( U(T) \) in Figure 3.7 which is based on the algorithm for computing \( \text{dbf}_T(t) \) in Figure 3.4 on page 30. Apart from changing initialization and update procedures of sets \( UT_k \) (replacing \( DT_k \)) accordingly, we run the outermost loop just for \( k = 1, \ldots, n \) and return the following:

\[
\max \left\{ \frac{e}{p} \mid \langle e, p, (u, v) \rangle \in \bigcup_{0 < k \leq n} UT_k \text{ and } u = v \right\}
\]

(If the set is empty, i.e., no cycles exist, we return 0 by defining \( \max \emptyset := 0 \).)

**function** compute-\( U(T) \):

1: \( UT_0 \leftarrow \{ \langle 0, 0, (v, v) \rangle \mid v \in V(T) \} \)

2: **for** \( k = 1 \) **to** \( n \) **do**

3: \( UT_k \leftarrow \emptyset \)

4: **for** all \( \langle e, p, (u, v) \rangle \in UT_{k-1} \) **do**

5: **for** all edges \( (v, v') \) in \( G(T) \) **do**

6: \( e' \leftarrow e + e(v) \)

7: \( p' \leftarrow p + p(v, v') \)

8: \( UT_k \leftarrow UT_k \cup \{ \langle e', p', (u, v') \rangle \} \)

9: **end** **for**

10: **end** **for**

11: **end** **for**

12: **return** \( \max \left\{ \frac{e}{p} \mid \langle e, p, (u, v) \rangle \in \bigcup_{0 < k \leq n} UT_k \text{ and } u = v \right\} \)

**Figure 3.7.** An iterative algorithm for computing \( \text{dbf}_T(t) \).

This procedure generates utilization triples corresponding to all paths of length \( k \leq n \). These must include all simple paths and therefore all simple cycles. Their utilizations can be directly derived from the utilization triples, and the maximum is returned. Finally, the procedure clearly has pseudo-polynomial run-time complexity.
3.2.5 Optimizations

We presented the schedulability algorithm in this section with extra focus on readability and correctness proofs. However, implementations may gain a lot of efficiency by applying a few optimizations.

**Computation Order**

The algorithm in Figure 3.4 computes the demand-bound function for a particular task and a particular interval size $t$. This can be improved in two ways.

First, we note that the feasibility test needs to call this function again and again for increasing $t$ until either the task set is shown to be infeasible by discovering a counterexample to Condition (3.1), or until all $t < D$ have been tested. This leads to a lot of duplicated effort: for each $t$, all demand triples are being recreated. In an efficient implementation, one would only create demand triples once up to $D$ and derive the demand-bound function for all $t < D$ from that set.

Second, we observed in experiments based on randomly generated tasks an asymmetry between feasible and infeasible task sets. A feasible task set usually has rather low utilization, leading to a small $D$ and therefore a very fast test. On the other hand, task sets with very high utilization, usually infeasible, will have a large $D$. However, with very high utilization, task sets tend to be clearly infeasible and counterexamples $t_f$ to Condition (3.1) usually occur rather early in that case. Computing all demand triples for all tasks up to $D$ would be rather wasteful. A very effective optimization is therefore to compute demand triples with increasing $d$ iteratively and checking Condition (3.1) along the way. This trick does not slow down the method for feasible task sets, but has a significant speed-up for infeasible ones. The longest run-time is observed for task sets close to the phase-change between feasible and infeasible task sets, that is, with a large $D$ but either a large $t_f$ or none at all. We discuss this issue further in Chapter 8 when evaluating our implementation.

**Critical Paths**

Another observation is that the sets $DT_k$, containing all demand triples representing $k$-paths, do not need to be kept separated, but can rather be implemented as a common store of all already visited demand triples. This idea makes it possible to detect that a demand triple has already been considered earlier (possibly representing a shorter path). More importantly, it also enables a more sophisticated optimization as follows. The key idea is that not only can a demand triple be abandoned if it itself has already been considered before, but even if a different, dominating one has already been considered. By this we mean that the new demand triple would not contribute new information to the calculation of $dbf_T(t)$, neither by itself nor by any future extension. We illustrate this idea with the following example.
Example 3.2.17. In our running example task T from Figure 2.2 on page 17, consider again path \( \pi = (v_4, v_2, v_3) \) from Example 3.1.3. We know that it corresponds to demand triple \( \langle 9, 43, v_3 \rangle \) as we calculated before. Another path also ending in \( v_3 \) is \( \pi' = (v_3, v_1, v_2, v_3) \) and we find that it corresponds to demand triple \( \langle 9, 44, v_3 \rangle \).

Intuitively, \( \pi \) generates the same execution demand as \( \pi' \) but during a shorter time interval. Since they both end in the same vertex, they have the same extensions, and thus all paths prefixed with \( \pi \) will always dominate those prefixed with \( \pi' \). Clearly, \( \pi' \) is not critical for further considerations. Thus, even though the demand triples are not equal, \( \langle 9, 44, v_3 \rangle \) can be discarded directly. Each demand triple that later on would be based on it would always have a counterpart based on \( \langle 9, 43, v_3 \rangle \) with a strictly smaller deadline. □

In general, domination can be defined for paths, but we skip directly to demand triples and define this concept as follows.

Definition 3.2.18 (Domination). Let \( \xi_1 = \langle e_1, d_1, u \rangle \) and \( \xi_2 = \langle e_2, d_2, v \rangle \) be two demand triples. We say that \( \xi_1 \) dominates \( \xi_2 \), written \( \xi_1 \succeq \xi_2 \), if

\[
e_1 \geq e_2, \quad d_1 \leq d_2, \quad \text{and} \quad u = v.
\]

We say that \( \xi_1 \) strictly dominates \( \xi_2 \), written \( \xi_1 \succ \xi_2 \), if \( \xi_1 \succeq \xi_2 \) and \( \xi_1 \neq \xi_2 \). □

Definition 3.2.19 (Critical Paths). A finite path \( \pi \) in \( G(T) \) corresponding to a demand triple \( \xi \) is critical, if there is no path \( \pi' \) corresponding to a demand triple \( \xi' \) with \( \xi' \succ \xi \). We also say that \( \xi \) is critical. □

The proposed optimization is to discard all demand triples which clearly are not critical when the procedure already has discovered a dominating one. By keeping demand triples per end vertex sorted by deadline, this optimization can be implemented very efficiently, resulting in a highly generalized variant of the \( \text{dbf} \) procedure presented in [16] for DAGs. Our prototype implementation showed a huge performance improvement using this optimization (from several minutes for large tasks down to a few seconds), since calculating demand pairs for \( \text{dbf}(t) \) becomes roughly linear in \( t \) instead of being quadratic. Further, correctness is guaranteed by the following lemma. It directly implies that demand triples corresponding to non-critical paths can be discarded, since any extension will be non-critical as well.

Lemma 3.2.20 (Optimal Substructure). For each critical path \( \pi = (\pi_0, \ldots, \pi_l) \), all prefixes \( \pi' = (\pi_0, \ldots, \pi_j), j \leq l \) are critical as well. □

Proof. The intuition for this lemma is as follows. If some prefix \( \pi^{\text{prefix}} \) of a critical path \( \pi \) is not critical, then \( \pi^{\text{prefix}} \) is dominated by another path \( \pi' \). This \( \pi' \)
could replace $\pi^{pfx}$ in $\pi$, resulting in a path that dominates $\pi$ (and thus contradicts that $\pi$ is critical).

Formally, we show the lemma by induction on the path length $l$.

$l = 0$: A 0-path has no prefixes except itself.

$l - 1 \Rightarrow l$: Given a critical path $\pi = (\pi_0, \ldots, \pi_l)$, assume that its $(l-1)$-prefix $\pi^{pfx} = (\pi_0, \ldots, \pi_{l-1})$ is not critical. Then there must be a $\pi' = (\pi'_0, \ldots, \pi'_k)$ such that

$$\langle e(\pi'), d(\pi'), \pi'_k \rangle \succ \langle e(\pi^{pfx}), d(\pi^{pfx}), \pi_{l-1} \rangle.$$

In particular, $\pi'_k = \pi_{l-1}$. We can now use $\pi'$ to replace the prefix $\pi^{pfx}$ of $\pi$. Consider the new path $\pi'' = (\pi'_0, \ldots, \pi'_k, \pi_l)$.

We have that

$$e(\pi'') = e(\pi') + e(\pi_l) \geq e(\pi^{pfx}) + e(\pi_l) = e(\pi),$$

$$d(\pi'') = d(\pi') - d(\pi'_k) + p(\pi'_k, \pi_l) + d(\pi_l)$$

$$\leq d(\pi^{pfx}) - d(\pi'_k) + p(\pi'_k, \pi_l) + d(\pi_l)$$

$$= d(\pi^{pfx}) - d(\pi_{l-1}) + p(\pi_{l-1}, \pi_l) + d(\pi_l)$$

$$= d(\pi).$$

Consequently, the new path $\pi''$ dominates $\pi$, i.e.,

$$\langle e(\pi''), d(\pi''), \pi_l \rangle \succ \langle e(\pi), d(\pi), \pi_l \rangle.$$

This contradicts that $\pi$ is critical. Thus, $\pi^{pfx}$ must be critical as well. By induction hypothesis, all prefixes of $\pi^{pfx}$ are also critical, which completes the proof for all prefixes of $\pi$.

3.3 Arbitrary Deadlines

The algorithm for computing the demand-bound function presented above assumes constrained deadlines, or at least monotonic absolute deadlines, cf. page 17. We now relax the deadline constraints, allowing $d(u) > p(u, v) + d(v)$ for the edges $(u, v)$ in $G(T)$. We assume no additional restrictions, i.e., jobs may execute as soon as they are released. In particular, they do not have to wait until preceding jobs of the same task are finished.

In this setting, the Feasibility Theorem 3.1.4 on page 25 still applies, so the general approach of checking feasibility using the demand-bound function is applicable. However, the arbitrary deadlines impose an additional analysis challenge, illustrated by the following example.
Example 3.3.1. Consider the DRT task in Figure 3.8. Since $d(v_2) > p(v_2, v_3) + d(v_3)$, the (absolute) deadline of a job may be after the deadline of its succeeding job. An example of this is the job sequence that is sketched in Figure 3.8 as well.

In this example, path $(v_1, v_2, v_3)$ corresponds to two demand pairs:

- $\langle 7, 16 \rangle$, because within 16 time units, all three jobs are released, but only the first and third are having their deadlines. Together, they have an execution demand of 7 time units.
- $\langle 8, 18 \rangle$, since with additional 2 time units, also the second job has its deadline within the considered interval. Thus, all three job execution demands have to be added, resulting in 8 time units.

Clearly, both demand pairs are relevant for the demand-bound function, since they do not dominate each other in the sense defined in Section 3.2.2 for paths and demand triples.

We see from this example that our procedure from the previous section needs to be extended in order to deal with arbitrary deadlines. In particular, we can no longer implicitly assume that the last job of each finite job sequence also has the latest deadline.

Marked Paths

For dealing with the general case, we first augment the notion of a path. The extra information we add should represent the subset of its vertices that we consider when calculating execution demand and deadline for a demand pair.
Definition 3.3.2 (Marked Path). For a path \( \pi = (\pi_0, \ldots, \pi_l) \) in \( G(T) \), we call a function \( \alpha \) a marking of its vertices, if \( \alpha(0) = \bullet \) and \( \alpha(j) \in \{\bullet, \circ\} \) for all \( j > 0 \). A path \( \pi \) together with its marking \( \alpha \) is a marked path \( \hat{\pi} := (\pi, \alpha) \). □

As shorthand notation, we attach to each vertex its individual marking when writing the path. For example, the path corresponding to the first demand pair \( \langle 7, 16 \rangle \) from Example 3.3.1 can be written as \( (v_1^\bullet, v_2^\circ, v_3^\bullet) \), since \( v_2 \) was not considered for \( \langle 7, 16 \rangle \) and is thus marked with \( \circ \).

In order to determine the corresponding demand pair for a marked path \( \hat{\pi} \), we now only consider its \( \bullet \)-marked vertices\(^3\). The execution demand \( e(\hat{\pi}) \) is the sum of their individual execution demands. The deadline \( d(\hat{\pi}) \) is derived from the latest deadline among all \( \bullet \)-marked vertices. Formally, we define both as follows.

Definition 3.3.3. For a marked path \( \hat{\pi} \), let \( \alpha^{-1}(\bullet) \) denote the set of positions of the \( \bullet \)-marked vertices in \( \hat{\pi} \). Using this, we define:

\[
e(\hat{\pi}) := \sum_{j \in \alpha^{-1}(\bullet)} e(\pi_j),
\]

\[
d(\hat{\pi}) := \max_{j \in \alpha^{-1}(\bullet)} \left\{ \sum_{i=0}^{j-1} p(\pi_i, \pi_{i+1}) + d(\pi_j) \right\}.
\]

As before, \( \langle e(\hat{\pi}), d(\hat{\pi}) \rangle \) denotes a demand pair. □

With these notions, we now show how to extend the procedures from the previous section for calculating both \( \text{dbf}(t) \) and the bound for \( t_f \).

Calculating Demand Pairs

The additional challenge when calculating demand pairs is the problem of dealing with different markings of the same path. As in Section 3.2, we would like to apply the iterative generation using demand triples by having each demand triple now correspond to a marked path. Consider a path \( \pi = (\pi_0^\bullet, \pi_1^\bullet, \pi_2^\circ, \pi_3^\bullet, \pi_4^\circ) \) which we want to extend with another vertex \( v \). There are two possibilities, \( \pi' = (\pi_0^\bullet, \pi_1^\bullet, \pi_2^\circ, \pi_3^\circ, v^\bullet) \) and \( \pi'' = (\pi_0^\bullet, \pi_1^\circ, \pi_2^\circ, \pi_3^\circ, \pi_4^\circ, v^\circ) \). Thus, given a demand triple \( \xi \) for \( \pi \), we have to generate one for \( \pi' \) and one for \( \pi'' \). However, this cannot be done directly from \( \xi \). It includes only a deadline (of either \( \pi_0 \) or \( \pi_2 \), whichever comes later). What is lost is the information about when the job triggered by \( \pi_4 \) is released, but we need this information in order to be able to generate the demand triple for \( \pi' \).

Our solution for this is to extend the abstraction to demand quadruples.

---

\(^3\)The initial vertex is always \( \bullet \)-marked, which is a technicality for simplification of presentation without loss of generality.
Definition 3.3.4 (Demand Quadruple). For a marked path \( \hat{\pi} = (\pi_0, \ldots, \pi_l) \) in \( G(T) \), we define its duration as

\[
p(\hat{\pi}) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}).
\]

With \( e(\hat{\pi}) \) and \( d(\hat{\pi}) \) from Definition 3.3.3, we call \( \langle e(\hat{\pi}), d(\hat{\pi}), p(\hat{\pi}), \pi_l \rangle \) a demand quadruple. \( \square \)

Clearly, there are only pseudo-polynomially many demand quadruples with \( e, d, p \leq D \) for any bound \( D \). Thus, we can adjust the iterative algorithm in Figure 3.4 on page 30 from before to use the demand quadruple abstraction in order to obtain a pseudo-polynomial procedure for calculating \( dbf_T(t) \). The resulting algorithm is shown in Figure 3.9.

```
function compute-dbft(T):
    1: \( DQ_0 \leftarrow \{ \langle e(v), d(v), 0, v \rangle \mid v \in V(T) \} \)
    2: for \( k = 1 \) to \( t \) do
    3: \( DQ_k \leftarrow \emptyset \)
    4: for all \( \langle e, d, p, u \rangle \in DQ_{k-1} \) do
    5:     for all edges \( (u, v) \) in \( G(T) \) do
    6:         \( e' \leftarrow e + e(v) \)
    7:         \( d' \leftarrow \max(d, p + p(u, v) + d(v)) \)
    8:         \( p' \leftarrow p + p(u, v) \)
    9:         \( e'' \leftarrow e \)
   10:         \( d'' \leftarrow d \)
   11:         \( p'' \leftarrow p + p(u, v) \)
   12:         if \( d' \leq t \) then
   13:             \( DQ_k \leftarrow DQ_k \cup \{ \langle e', d', p', v \rangle \} \)
   14:         end if
   15:         if \( d'' \leq t \) then
   16:             \( DQ_k \leftarrow DQ_k \cup \{ \langle e'', d'', p'', v \rangle \} \)
   17:         end if
   18:     end for
   19: end for
   20: return \( \max \{ e \mid \langle e, d, p, v \rangle \in \bigcup_{k \leq t} DQ_k \} \)
```

Figure 3.9. An iterative algorithm for computing \( dbf_T(t) \) in the case of arbitrary deadlines.

For each \( \langle e, d, p, u \rangle \) chosen in line 4 and edge \( (u, v) \) from line 5, we create two new demand quadruples: \( \langle e', d', p', v \rangle \) for extension with a \( \bullet \)-marked \( v \) and
\begin{equation*}
\langle e', d', p', v \rangle \text{ for a } \circ\text{-marked } v. \text{ Their calculation is straightforward:}
\end{equation*}
\begin{align*}
e' &= e + e(v) & e'' &= e \\
d' &= \max(d, p + p(u, v) + d(v)) & d'' &= d \\
p' &= p + p(u, v) & p'' &= p + p(u, v)
\end{align*}

The first tuple \( \langle e', d', p', v \rangle \) extends \( \langle e, d, p, u \rangle \) with a \( \bullet \)-marked \( v \), so \( e' \) needs to include \( e(v) \) and \( d' \) needs to be extended to include the deadline of \( v \) as well. It should not be smaller than \( d \) already was, since an earlier marked vertex may already have a later deadline. Finally, \( p' \) (as well as \( p'' \)) need to include the new edge. The second tuple \( \langle e'', d'', p'', v \rangle \) extends \( \langle e, d, p, u \rangle \) with a \( \circ \)-marked \( v \), so \( e'' \) and \( d'' \) should just copy the old values of \( e \) and \( d \).

It can be easily seen that with these changes, the procedure is still correct. Further, with a suitable domination relation \( \succ \), all optimizations discussed in Section 3.2.5 can be applied for efficient implementations.

**Example 3.3.5.** Consider the DRT task from Figure 3.8 on page 43. Vertex \( v_1 \) can be interpreted as a marked 0-path \( (v_1^\bullet) \) and thus corresponds to demand quadruple \( \langle 5, 7, 0, v_1 \rangle \). We can extend it with \( v_2 \), which gives the possibilities \( \langle 6, 18, 8, v_2 \rangle \) and \( \langle 5, 7, 8, v_2 \rangle \), corresponding to marked paths \( (v_1^\bullet, v_2^\bullet) \) and \( (v_1^\bullet, v_2^\circ) \), respectively. Note that if we extend both of these with a \( \bullet \)-marked \( v_3 \), we get the demand quadruples \( \langle 8, 18, 11, v_3 \rangle \) and \( \langle 7, 16, 11, v_3 \rangle \) from which both demand pairs in Example 3.3.1 are derived. \( \square \)

**Calculating the Bound**

The bound for \( t_f \) that we derived in Section 3.2.3 turns out to hold also for the case of arbitrary deadlines. A first observation is that the linear bound in Lemma 3.2.10 on page 34 does not depend on any deadlines.

More formally, given a task \( T \), we transform it into a task \( T' \) where we reduce the deadlines such that \( T' \) now has monotonic absolute deadlines\(^4\). Clearly, \( dbf_T(t) \leq dbf_{T'}(t) \) for all \( t \). Further, we know that \( U(T) = U(T') \) and \( e^{\text{sum}}(T) = e^{\text{sum}}(T') \) since both their definitions are independent of deadlines. It follows directly that Lemma 3.2.10 on page 34 about a linear bound for the demand-bound function holds even for arbitrary-deadline task sets. Finally, this implies that also Lemma 3.2.11 on page 36 about the maximal value of a counterexample \( t_f \) must hold. Therefore, we can use the same bound \( D \) as before.

We summarize all insights from this section in the following theorem.

**Theorem 3.3.6.** For a DRT task set \( \tau \) with arbitrary deadlines and \( U(\tau) \leq c \) for some constant \( c < 1 \), feasibility can be decided in pseudo-polynomial time. \( \square \)

\(^4\)This may lead to \( e(v) > d(v) \) for some vertices \( v \) in \( T' \), but does not cause any technical difficulty, since we do not demand equivalence of \( T \) and \( T' \) in terms of schedulability.
4. Tractability Borderline

The feasibility analysis procedure in the previous chapter shows that pseudo-polynomial run-time complexity can be achieved for the analysis of DRT task systems. Since the model is a generalization of a hierarchy of many earlier models, cf. Chapter 9, a natural question to ask is how much the model can be generalized until pseudo-polynomial feasibility analysis is not possible anymore.

We answer the question in this chapter by extending the basic DRT model with global inter-release separation constraints between non-adjacent job releases. This extension makes the model strictly more expressive and demonstrates the borderline between tractable (that is, with a pseudo-polynomial test) and intractable (strongly coNP-hard). If the number of constraints is bounded by a constant, the feasibility problem can be reduced to an analysis for DRT tasks from the previous chapter. If the number is not bounded, we demonstrate that the problem is strongly coNP-hard.

4.1 Global Constraints

We extend the DRT task model with global constraints which allow to specify a minimal amount of time that has to pass between job releases which do not correspond to adjacent jobs, i.e., the corresponding vertices do not need to be connected by an edge. We present syntax and semantics of the extended task model.

4.1.1 Syntax

An extended digraph real-time (EDRT) task system $\tau = \{T_1, \ldots, T_N\}$ is a DRT task system, cf. Section 2.2.1 on page 16, with the following additional component. A constraint set $C(T) = \{(\text{from}_1, \text{to}_1, \gamma_1), \ldots, (\text{from}_k, \text{to}_k, \gamma_k)\}$ for each task $T \in \tau$ with $\text{from}_i, \text{to}_i \in V(T)$ and $\gamma_i \in \mathbb{N}$ contains $k$ global minimum inter-release separation constraints. Each constraint $(\text{from}_i, \text{to}_i, \gamma_i) \in C(T)$ expresses that between the visits of vertices $\text{from}_i$ and $\text{to}_i$, at least $\gamma_i$ time units must pass. For a fixed constant $k$, we call a task $T$ with $k = \|C(T)\|$ a $k$-EDRT task.

Constraints in $C(T)$ can involve any pair of vertices, they can be self-loops, they can be connected or unconnected vertices, the constraints could even be 0. (Some of these combinations would only make very limited sense.)
Figure 4.1. An example EDRT task containing five different types of jobs and two additional constraints \((v_4, v_2, 6)\) and \((v_3, v_3, 9)\), denoted with dashed arrows. Note that these dashed arrows do not represent edges that can be taken. They only denote additional timing constraints.

Example 4.1.1. Figure 4.1 shows an example of a 2-EDRT task with constrained deadlines. We will use it as a running example throughout the rest of this chapter.

4.1.2 Semantics

As for DRT tasks in Section 2.2.2, we define the semantics of EDRT task systems \(\tau\) in terms of generated job sequences. As before, these correspond to paths through the graphs \(G(T)\) of \(T \in \tau\). To explicitly denote necessary waiting times at the vertices, we extend paths in \(G(T)\) with delay information.

**Definition 4.1.2 (Timed Path).** For a path \(\pi = (\pi_0, \ldots, \pi_l)\) in \(G(T)\), a timed path \(\tilde{\pi} = (\pi_0, \delta_0, \pi_1, \delta_1, \ldots, \delta_{l-1}, \pi_l)\) is derived by interleaving vertices with waiting times \(\delta_i \in \mathbb{R}_{\geq 0}\). We call a timed path \(\tilde{\pi}\)

- **legal**, if the waiting times are consistent with the given inter-release separation constraints, i.e., if for \(i < j \leq l\):
  
  1. \(\delta_i \geq p(\pi_i, \pi_{i+1})\), and
  2. \(\delta_i + \delta_{i+1} + \ldots + \delta_{j-1} \geq \gamma\) for all \((\pi_i, \pi_j, \gamma) \in C(T)\).

- **urgent**, if all \(\delta_i\) are minimal so that is still \(\tilde{\pi}\) legal, i.e., for all prefixes \((\pi_0, \delta_0, \ldots, \delta_{j-1}, \pi_j)\) of \(\tilde{\pi}\) and all \(\varepsilon > 0\), the timed path \((\pi_0, \delta_0, \ldots, \delta_{j-1} - \varepsilon, \pi_j)\) is not legal.
This extends to infinite paths in a natural way. Note that legal timed paths may contain non-integral delays $\delta_i$, but for urgent timed paths, all $\delta_i$ are necessarily integers\(^1\). Using timed paths, the following definition is an extension of Definition 2.2.2 on page 18 for DRT tasks.

**Definition 4.1.3 (EDRT Generated Job Sequence).** For an EDRT task $T$, a job sequence $\sigma = [(R_0, e_0, \pi_0), (R_1, e_1, \pi_1), \ldots]$ is generated by $T$, if there are delays $\delta_0, \delta_1, \ldots$ such that $\pi = (\pi_0, \delta_0, \pi_1, \delta_1, \ldots)$ is a legal timed path in $G(T)$ and for all $i \geq 0$:

1. $R_{i+1} - R_i = \delta_i$ and
2. $e_i \leq e(\pi_i)$.

We denote the set of all job sequences generated by $T$ with $[T]$. □

This is a generalization of Definition 2.2.2 on page 18 where $\delta_i$ is any real number which is at least $p(\pi_i, \pi_{i+1})$. However, in the case of EDRT tasks, $\delta_i$ may have to be larger than $p(\pi_i, \pi_{i+1})$, since additional waiting time caused by global constraints has to be taken into account as well.

**Example 4.1.4.** For the example task $T$ in Figure 4.1, consider the job sequence $\sigma = [(2.3, 1, v_4), (5.1, 1, v_5), (8.9, 2, v_2)]$. It is generated by $T$, since it corresponds to timed path $\tilde{\pi} = (v_4, 2.8, v_5, 3.8, v_2)$ which is legal. If we reduce the second delay in $\tilde{\pi}$ to get $\tilde{\pi}' = (v_4, 2.8, v_5, 3, v_2)$, the result would not be legal: $v_2$ is visited 5.8 time units after $v_4$ which makes $\tilde{\pi}'$ violate the constraint $(v_4, v_2, 6)$.

Note that $\tilde{\pi}$ is legal, but not urgent. An urgent timed path involving the same sequence of vertices is $\tilde{\pi}'' = (v_4, 2, v_5, 4, v_2)$. In fact, $\tilde{\pi}''$ is the unique urgent timed path along these vertices since each path through $G(T)$ corresponds uniquely to an urgent timed path. □

As for DRT tasks, job sequences generated by task systems $\tau$, written $[\tau]$, are compositions of job sequences by all tasks $T$, written $[T]$, just as before (cf. Definition 2.2.4 on page 19).

### 4.2 Analysis for $k$-Bounded Case

We now develop a pseudo-polynomial feasibility analysis algorithm for $k$-EDRT task systems where $k$ is a constant. The analysis is based on the feasibility test presented in Chapter 3. This is possible since the Feasibility Theo-\(^1\)Otherwise, the fractional part of the first non-integral delay could be moved to the next delay. This preserves legality of the path, but shortens the accumulated duration of the prefix, showing that the original path was not urgent.
rem 3.1.4 on page 25 also applies to EDRT task systems. Recall that according to this theorem, feasibility is equivalent to the following condition:

$$\forall t \geq 0 : df(t) \leq t.$$ 

However, the method presented in Chapter 3 for testing this condition cannot be applied directly to EDRT task systems. Recall that the key idea for a feasibility test for DRT task systems is to use the demand triple path abstraction in order to efficiently collect demand information from the involved digraphs $G(T)$ for each task $T$. A demand triple $\langle e(\pi), d(\pi), \pi_t \rangle$ abstracts a path $\pi$ with its accumulated execution time, relative deadline and last vertex $\pi_t$. This is the smallest abstraction with the following properties:

1. It contains the demand pair information necessary to compute the demand-bound function.
2. It can be extended to represent longer and longer paths, i.e., paths containing $\pi$ as a prefix.

For an EDRT task, the demand triple abstraction is not sufficient to preserve the second property, i.e., extensibility. The problem becomes clearer when described using (urgent) timed paths. Each path $\pi = (\pi_0, \pi_1, \ldots, \pi_l)$ corresponds to a unique urgent timed path $\tilde{\pi} = (\pi_0, \delta_0, \pi_1, \delta_1, \ldots, \delta_{l-1}, \pi_l)$. In the DRT model, the delays of urgent timed paths are equal to the edge labels: $\delta_i = p(\pi_i, \pi_{i+1})$. This makes it straightforward to extend $\tilde{\pi}$ with a vertex $v$. The extended urgent timed path is just $(\pi_0, \delta_0, \ldots, \pi_l, \delta_l, v)$ with $\delta_l = p(\pi_l, v)$. This is why the demand triple abstraction works well for DRT tasks, since it only needs to record the last vertex of the abstracted paths.

However, in the EDRT model with additional constraints from $C(T)$, additional waiting may be required, if $v$ is the $io_i$ vertex of some constraint $(from_i, io_i, \gamma_i) \in C(T)$. We call such a constraint active. Therefore, when extending $\tilde{\pi}$ to $(\pi_0, \delta_0, \ldots, \pi_l, \delta_l, v)$, the new value of $\delta_l$ may depend on earlier vertices visited in $\tilde{\pi}$ and their delays. In other words, since the demand triple abstraction only records the last vertex, it “forgets” information about the active constraints.

**Example 4.2.1.** To illustrate this problem, consider the urgent timed path $\tilde{\pi}'' = (v_4, 2, v_5, 4, v_2)$ from Example 4.1.4 on page 49 for the example task $T$ in Figure 4.1 on page 49. The graph exploration technique from Chapter 3 would start with a demand triple $\xi^{(1)} = \langle 1, 2, v_4 \rangle$ for 0-path $\tilde{\pi}^{(1)} = (v_4)$ and extend it to $\xi^{(2)} = \langle 2, 4, v_5 \rangle$ as an abstraction of path $\tilde{\pi}^{(2)} = (v_4, 2, v_5)$. Clearly, this demand triple $\xi^{(2)}$ (in contrast to the timed path $\tilde{\pi}^{(2)}$ that it abstracts) lost the information that constraint $(v_4, v_2, 6)$ is active at $v_5$. Thus, we cannot derive from $\xi^{(2)}$ that a delay of at least 4 time units is necessary before visiting $v_2$.

A similar problem arises for determining a task’s utilization. This is done in Chapter 3 by finding simple cycles in $G(T)$. Consider cycle $(v_5, v_2, v_3, v_5)$
in \( G(T) \). Because of the constraint on revisiting \( v_3 \) earliest after 9 time units, the duration of this cycle is actually 9 time units, giving it a density of \( 5/9 \). (That is, accumulated execution time for all vertices but the last, divided by the path’s duration.) The other simple cycle \((v_5, v_4, v_5)\) has an even lower density of \( 1/2 \). However, the real utilization of \( T \) is in fact \( 7/12 \), demonstrated via cycle \((v_5, v_2, v_3, v_5, v_4, v_5)\) with a density of \( 7/12 \). Note that this cycle is not simple and not easy to find.

In summary, we are facing two challenges when adapting the feasibility test to the \( k \)-EDRT setting:

1. While traversing \( G(T) \) using the demand triple abstraction, we must keep information about the state of all \( k \) constraints, i.e., to what extent they are active.

2. The utilization of a task cannot be determined easily by just considering simple cycles in \( G(T) \). The additional constraints must be honored and a “most dense” timed cycle may not be simple.

We solve both by translating each given \( k \)-EDRT task \( T \) into an equivalent plain DRT task \( T' \). The key idea is to store information about the constraints of the original task in the vertices of the new task while adjusting the edges accordingly. For each constraint \((\text{from}_i, \text{to}_i, \gamma_i)\), we keep a countdown starting at \( \gamma_i \) in all vertices. It records how many time units at least passed since \( \text{from}_i \) has been visited the last time. Consequently, vertex \( \text{to}_i \) is only allowed to be visited when the corresponding countdown is 0, potentially imposing an additional waiting penalty. These additional delays are considered when labeling the edges of \( T' \), making the edge labels potentially larger than the corresponding edge labels of \( T \). After the translation of all tasks in the \( k \)-EDRT model \( \tau \), we can run the analysis method from Chapter 3 on the newly created DRT model \( \tau' \) in order to solve the feasibility problem. Thus, the task transformation is the main focus for the rest of this section.

Note that the countdowns can be restricted to integer values, since we are only interested in urgent timed paths through \( G(T) \), which in turn only contain integer delays. However, even with integers, the set of vertices of \( G(T') \) may grow rapidly in size compared to \( G(T) \), although this growth is polynomially bounded in the constraint values \( \gamma \). We introduce some optimizations in Section 4.2.3 that greatly reduce the number of vertices in \( T' \) and the actual overhead during graph traversal.

### 4.2.1 Task Transformation Procedure

We start by illustrating the task transformation using a very basic example.

---

51
Example 4.2.2. Consider the example in Figure 4.2. The shown task $T$ has three vertices with the constraint $(v_1, v_3, 5)$. Since there is one constraint, we extend each vertex with one countdown:

- **Vertex** $v_1$ is extended to $(v_1, 5)$ in $T'$ since it is the starting vertex of the constraint. The constraint value is 5, so the countdown gets the value 5.

- **From** $(v_1, 5)$, we create an edge to $(v_2, 3)$ since there is an edge $(v_1, v_2)$ in $T$ with label $p(v_1, v_2) = 2$. The constraint does not involve $v_2$, so the countdown is just decreased by 2, which is also the edge label.

- **From** $(v_2, 3)$, we want to create an edge to a vertex involving $v_3$, since there is an edge $(v_2, v_3)$ in $T$. However, $v_3$ is the target vertex of the given constraint. Thus, the countdown needs to decrease to 0, to ensure constraint satisfaction. Consequently, the edge goes to vertex $(v_3, 0)$ in $T'$. The edge label is 3 which is the required waiting time.

- **Finally**, if one starts at $v_2$, the constraint is not active, which we model by another vertex $(v_2, 0)$ in $T'$. Its countdown is 0 and thus allows visiting $(v_3, 0)$ after just $p(v_2, v_3) = 2$ time units.

Note that the actual transformed task $T'$ contains more vertices for $v_2$ and $v_3$ with other countdown values. They are however not essential and therefore left out for this example. □

We now give the full details of how to construct an equivalent plain DRT model $T'$ from a $k$-EDRT model $T$. We need to describe the set of vertices, their labels, the set of edges and their labels. Note that this is a theoretical description of the transformation and sufficient for the theoretical complexity result. We introduce some powerful optimizations in Section 4.2.3 for efficient implementations.
**Vertices.** For each vertex \( v \in V(T) \), we create vertices \((v, t)\) for \( G(T') \), where \( t = (t_1, \ldots, t_k) \) is a countdown vector. A priori, we do not know which of the possible values will actually be used, so we need to create vertices for all possible combinations of values for countdowns \( t_i \). A countdown \( t_i \) is associated with constraint \((\text{from}_i, \text{to}_i, \gamma_i)\). It has value \( \gamma_i \) if \( v \) is the starting vertex of the associated constraint, i.e., \( v = \text{from}_i \). This expresses that we want to record that \( T \) just visited \( \text{from}_i \) and at least \( \gamma_i \) time units must pass before \( \text{to}_i \) may be visited. Otherwise, \( t_i \) can have any integer value between 0 and \( \gamma_i \), since we do not know beforehand how long ago \( \text{from}_i \) has been visited last when visiting \( v \) in \( T \). All possible \((v, t)\) consistent with this description are being created. Formally, for each \( v \in V(T) \), we create as vertices for \( T' \) all \((v, t)\) such that:

\[
\forall i : \begin{cases} 
    t_i = \gamma_i & \text{if } v = \text{from}_i, \\
    t_i \in \{0, \ldots, \gamma_i\} & \text{otherwise.}
\end{cases}
\]

**Vertex labels.** Each new vertex \((v, t)\) in \( G(T') \) has the same label as vertex \( v \) in \( G(T) \), since it represents the release of jobs of the same type. Formally:

\[
\forall (v, t) \in V(T') : \begin{cases} 
    e((v, t)) := e(v), \\
    d((v, t)) := d(v).
\end{cases}
\]

**Edges.** Given two vertices \((u, s)\) and \((v, t)\) in \( G(T') \) we need to decide whether there should be an edge between them in \( G(T') \). For each combination of \( u, s \) and \( v \) with \((u, v) \in E(T)\), there will be exactly one countdown vector \( t \) for which we create such an edge:

1. \( t_i = \gamma_i \) for all \( i \) such that \( v = \text{from}_i \), expressing that \( v \) is resetting a countdown since it is the starting vertex of the associated constraint.

2. Otherwise, \( t_i \) is \( s_i \) decremented by some waiting time. For all \( i \), the countdowns are decremented by the same amount of some time \( \delta \). Countdowns below 0 are set to 0.

3. The waiting time \( \delta \) is at least \( p(u, v) \) and also not smaller than any \( s_i \) for all \( i \) such that \( v = \text{to}_i \). This expresses that all constraints involving \( v \) as target vertex need to be satisfied and the waiting time sufficiently large to guarantee that.

Formally, we first calculate the waiting time \( \delta \):

\[
\delta(u, s, v) := \max \{ p(u, v) \} \cup \{ s_i \mid v = \text{to}_i \}_{i=1}^k.
\]
Second, the following must hold for the new countdown vector \( t \):

\[
\forall i : t_i = \begin{cases} 
\gamma_i & \text{if } v = \text{from}_i, \\
\max (0, s_i - \delta(u, s, v)) & \text{otherwise.}
\end{cases}
\]

**Edge labels.** For an edge from \((u, s)\) to \((v, t)\), the delay is just the \( \delta \) we computed above.

\[
p((u, s), (v, t)) := \delta(u, s, v).
\]

It’s important to note that the countdown vector does *not* represent dynamic information during “run-time”, but is a rather static property of the vertices in \( T' \). To illustrate this, consider again the task from Figure 4.2. In the transformed task in Figure 4.2(b), assume a timed path starts in \((v_1, 5)\). Assume further that the system now waits for more than the minimal time, e.g., 3 instead of 2 time units. When moving to the successor vertex representing \( v_2 \), we *still* arrive at \((v_2, 3)\). This is because the edge represents only the *minimal* possible waiting time. Therefore, the countdowns as a static part of the vertices only record the effect of the edges, not of the actual “run-time behavior”. Consequently, the system in this example is required to wait again for 3 time units (not just 2) before the next step, which would be to \((v_3, 0)\). In other words, the timed path \((v_1, 3, v_2, 2, v_3)\) which is legal in \( T \) does not have a corresponding timed path in \( T' \), i.e., with the same delays. However, this is not a problem, since it is not an *urgent* path. The objective of our presented task transformation is to preserve urgent paths of \( T \) (which indeed all have a counterpart in \( T' \)) since only these define the *dbf*.

**Example 4.2.3.** Consider again the running example task \( T \) from Figure 4.1 on page 48. After applying the described transformation to \( T \), we get an equivalent plain DRT task \( T' \) of which the essential parts are shown in Figure 4.3. In fact, after the vertex removal optimization discussed below in Section 4.2.3, the vertices from Figure 4.3 are the only remaining ones.

Note that the cycle \((v_5, v_2, v_3, v_5, v_4, v_5)\) in \( G(T) \) that we identified in Example 4.2.1 as the one with the highest density translates to a simple cycle in \( T' \).

\[ \square \]

### 4.2.2 Correctness

For the task transformation method from above, we show now its correctness and summarize the whole \( k \)-EDRT analysis method.

We use a central correctness lemma to prove that the demand-bound functions of the given \( k \)-EDRT task \( T \) and the transformed DRT task \( T' \) coincide. Intuitively, every urgent timed path in \( T \) has a legal corresponding path through \( T' \) by extending the vertices with appropriate countdown vectors. Further, all
urgent timed paths in $T'$ are legal in $T$ (after removing countdown vectors from the vertices) since the countdowns ensure satisfaction of all inter-release separation constraints.

**Lemma 4.2.4.** For a $k$-EDRT task $T$ and its transformation $T'$, their demand-bound functions coincide, i.e.,

$$\forall t \geq 0 : \text{dbf}_T(t) = \text{dbf}_{T'}(t).$$

**Proof.** In order to show that the demand-bound functions of $T$ and $T'$ are identical, it suffices to show the following two properties about urgent timed paths through their graphs:

1. An urgent timed path in $T$ has a corresponding legal timed path in $T'$.
2. An urgent timed path in $T'$ has a corresponding legal timed path in $T$.

Both together imply that there cannot be a $t$ for which either of $\text{dbf}_T(t)$ and $\text{dbf}_{T'}(t)$ is larger: both values correspond to some urgent timed path each. If each of them corresponds to a legal timed path in the other model, the other demand-bound function must be at least as large at that point.

For the first claim, let $\tilde{\pi} = (\pi_0, \delta_0, \ldots, \pi_l)$ be an urgent timed path in $T$. We iteratively construct a corresponding path $\tilde{\pi}' = ((\pi_0, t^{(0)}), \delta_0, \ldots, (\pi_l, t^{(l)}))$. All we need to do is to define all countdown vectors $t^{(l)}$ appropriately, since all other components are taken directly from $\tilde{\pi}$. We start with $t^{(0)}$, in which...
we set all countdowns as small as possible:

$$\forall m : t_m^{(0)} = \begin{cases} \gamma_m & \text{if } \pi_0 = \text{from}_m, \\ 0 & \text{otherwise.} \end{cases}$$

All succeeding $t^{(i)}$ are constructed by applying the rules presented in Section 4.2.1 for edge construction. Since $\tilde{\pi}$ is urgent in $T'$, all $\delta_i$ correspond exactly to the $\delta$ calculated in these rules. The resulting $\tilde{\pi}'$ must be a legal path in $T'$ since all $\delta_i$ are equal to the edge labels, by construction.

For the second claim, let $\tilde{\pi}' = ((\pi_0, t^{(0)}), \delta_0, \ldots, (\pi_l, t^{(l)}))$ be an urgent timed path in $T'$. We want to show that $\tilde{\pi} = (\pi_0, \delta_0, \ldots, \pi_l)$ is legal in $T$. From the construction of the waiting time $\delta$ in the transformation of $T$ to $T'$ it is clear that $\delta_i \geq p(\pi_i, \pi_{i+1})$ for all $i$. Further, for all constraints $(\text{from}_m, \text{to}_m, \gamma_m) \in C(T)$, any visit of $\text{to}_m$ in $\tilde{\pi}'$ cannot occur before the corresponding countdown is 0. Since the countdown is reset to $\gamma_m$ whenever $\text{from}_m$ is visited, the accumulated waiting time between $\text{from}_m$ and $\text{to}_m$ must be at least $\gamma_m$. Consequently, for any two constrained vertices in $\tilde{\pi}'$, i.e., $\pi_i = \text{from}_m$ and $\pi_j = \text{to}_m$, we have $\delta_i + \ldots + \delta_{j-1} \geq \gamma_m$. In summary, $\tilde{\pi}$ must be legal in $T$. ■

Given this lemma, the main theorem follows directly, since the result from Chapter 3 can be applied to the set of transformed tasks.

**Theorem 4.2.5.** For a $k$-EDRT task set $\tau$ with $U(\tau) \leq c_1$ for some constant $c_1 < 1$ and $k \leq c_2$ for some constant $c_2 \in \mathbb{N}$, feasibility can be decided in pseudo-polynomial time. □

**Proof.** Given $\tau$, we apply the described transformation to all tasks in order to obtain $\tau'$. Lemma 4.2.4 guarantees that their demand-bound functions coincide, which implies that their utilizations are also the same (Definition 3.2.8). Thus, we can apply the main result from Chapter 3, guaranteeing feasibility to be decidable for $\tau'$ (and therefore $\tau$) in pseudo-polynomial time, if the following two conditions hold:

1. The number of vertices in $\tau'$ as well as the values in $\tau'$ (vertex and edge labels) are pseudo-polynomially bounded in the description of $\tau$.

2. The transformation itself runs in pseudo-polynomial time.

For the first property, all labels are bounded by the old labels and the constraint values. Further, the number of new vertices per old vertex is bounded by $\prod_{i=1}^{k} (\gamma_i + 1)$, since this is the number of possible countdown vectors (including value 0 for each countdown). With $k$ being bounded by a constant, this is a polynomial in the values from $\tau$. The second property is trivially satisfied since creating each of these vertices and edges (and their labels) takes negligible time. ■

56
Note that we did not assume any constraints on deadlines. Thus, the result holds for task sets with arbitrary deadlines, since Lemma 4.2.4 only relies on the property that urgent timed paths are defining the demand-bound function. This is true for the case of arbitrary deadlines as well.

4.2.3 Optimizations

While the described method from above is sufficient to show the theoretical pseudo-polynomial complexity bound, there are a few ways to optimize implementations for higher efficiency. First, all optimizations discussed in Chapter 3 are applicable when analyzing the transformed task set. Further, we discuss three optimizations that are specific to the task transformation and may have drastic impact on analysis run-time: A refined domination relation, removal of unnecessary vertices and a countdown compression method.

Refined Domination Relation

In Chapter 3, a domination relation between demand triples is introduced as an optimization. A demand triple \((e, d, v)\) dominates \((e', d', v')\), if the abstracted paths can create a higher demand \((e \geq e')\) within a shorter interval \((d \leq d')\) and end in the same vertex \((v = v')\). The motivation is that in this case, \((e', d', v')\) does not contribute new information to the dbf calculation. Further, the same is true for all future extensions of \((e', d', v')\), since \((e, d, v)\) ends in the same vertex and can thus be extended in the exact same way. Consequently, \((e', d', v')\) does not need to be considered further and can be discarded.

This concept can be refined for the analysis of a translated \(T'\). We have domain specific information about the involved vertices: they represent countdown vectors. A countdown vector constrains future behavior of the path, since it may impose additional waiting times. Thus, if we have two vertices \((v, s)\) and \((v, t)\) in \(T'\) with \(\forall i: s_i \leq t_i\), all path extensions possible from \((v, t)\) are also possible from \((v, s)\), in potentially shorter time. This relation can be added to the domination relation in order to implement further discarding opportunities.

Example 4.2.6. Consider the following two paths through \(G(T')\) in Figure 4.3 on page 55, together with their demand triple abstractions:

\[
((v_5, (4, 3)), (v_4, (6, 1))) \rightsquigarrow (2, 4, (v_4, (6, 1)))
\]
\[
((v_5, (0, 7)), (v_4, (6, 5))) \rightsquigarrow (2, 4, (v_4, (6, 5)))
\]

Strictly speaking, both demand triples are involving different vertices from \(T'\), so no optimization from Chapter 3 could be applied for discarding one of them. However, we know that they include a representation of the same vertex in the original EDRT task \(T\) (Figure 4.1 on page 48). Further, comparing the countdown vector \((6, 1)\) to \((6, 5)\), we notice that the first one is less restrictive.
than the second one, since all countdowns are smaller or equal. Consequently, the second demand triple can be discarded.

**Vertex Removal**

For a second optimization, we note that the first optimization described above applies particularly to the initial demand triples. In our running example from Figure 4.1 on page 48, the transformation will produce, among others, the two vertices \((v_4, (6, 0))\) and \((v_4, (6, 8))\). (Only the first one is shown in Figure 4.3 on page 55.) They only differ in the countdown for the second constraint, and whatever timed path is possible from \((v_4, (6, 8))\) has a corresponding timed path starting in \((v_4, (6, 0))\), which is less restrictive, as discussed above. Thus, with the above optimization, already the initial demand triple containing \((v_4, (6, 8))\) can be discarded. We additionally notice that the vertex \((v_4, (6, 8))\) does not have any incoming edges, since these would reduce the second countdown by at least 2 (and it has a maximum of 9). This means that the graph exploration of the transformed task \(T'\) does not start at that vertex and actually will never visit it. Thus, we can remove that vertex from \(T'\) altogether without influencing the represented timed paths through \(T\) and thus the \(dbf\).

In general, we can remove from \(T'\) all vertices \((v, t)\) which do not have any incoming edges and for which \(t\) does not have the following shape:

\[
\forall i: t_i = \begin{cases} 
\gamma_i & \text{if } v = \text{from}_i, \\
0 & \text{otherwise}. 
\end{cases}
\]

This removal procedure can be repeated until no such vertex exists anymore. (Note that by removing vertices, other vertices may lose their incoming edges and be thus also eligible for removal.)

The described vertex removal procedure is quite effective. In the example from Figure 4.1 on page 48, the EDRT task has 5 vertices which results in 227 vertices after task transformation to a plain DRT task. However, after applying the described optimization, only 11 vertices remain (those shown in Figure 4.3 on page 55), resulting in an analysis speed-up of several orders of magnitude.

A different way of realizing this optimization is to create vertices only on the fly during analysis. This means that an implementation would not start the analysis by first creating all vertices and edges in memory, removing unnecessary ones afterwards. Instead, only the vertices with countdown vectors of the shape described above are created, i.e., one per vertex in the original EDRT model. As graph exploration proceeds, new vertices with updated countdowns are created. By doing so, only the necessary vertices are actually created and memory use may be reduced significantly.
Countdown Compression

For our third optimization, we note that a special situation occurs if constraints are located “far away” from each other in $T'$. Imagine a task with a rather big directed graph, where two vertices $u$ and $v$ are involved in constraints $(u, u, 10)$ and $(v, v, 10)$. Further, assume all timed paths between both vertices involve accumulated delays of more than 10. Clearly, at most one of the two constraints is active at any time. They do not overlap. Thus, their corresponding countdowns in $T'$ after the vertex removal described above are never non-zero at the same time.

In other words, their countdown could be re-used and thus we could compress the countdown vector. In the simple example of the two constraints $(u, u, 10)$ and $(v, v, 10)$, we would just use one countdown and an additional bit indicating which of the two constraints is being counted. In fact, if $T$ contains more than $k$ constraints, e.g., linearly many, it could still be analyzed by the presented method for $k$-EDRT, by using the sketched compression optimization. Consequently, Theorem 4.2.5 can be generalized to EDRT task sets with up to $k$ non-overlapping constraints. Note that the non-overlapping property can be verified by a simple graph traversal.

4.3 Hardness for Unbounded Case

While the previous section presents a pseudo-polynomial feasibility analysis for $k$-EDRT tasks, we show now that for unbounded $k$, the problem is strongly coNP-hard. This implies that a pseudo-polynomial analysis procedure cannot exist, assuming $P \neq NP$.

4.3.1 Reduction Framework

For our hardness proof, we use the standard notion of a polynomial-time many-one reduction [25], also called “Karp reduction”. For two decision problems $A$ and $B$, a Karp reduction from $A$ to $B$ is a polynomial-time algorithm that transforms instances of $A$ into instances of $B$ while preserving whether the instance is a positive or negative one. In that case, $A$ is reduced to $B$, written $A \leq_{\text{pol}} B$. This means that a polynomial-time algorithm solving $B$ can be used to solve instances of $A$ by first applying the reduction. Further, if $A$ is an $NP$-hard or coNP-hard problem, then $B$ must also be $NP$-hard or coNP-hard, respectively.

A problem can be solved in pseudo-polynomial time, if there is a polynomial time algorithm deciding membership for an instance representation where all values are encoded in unary. This is equivalent to the requirement that polynomial bounds exist for all values and is sometimes a practical requirement: not just the number of input objects to an algorithm should determine its run-
time, even their values are to be considered. Note that the run-time of all algorithms presented in the current and the previous chapter is pseudo-polynomial. Finally, a problem is \textit{NP-hard in the strong sense} if it stays \textit{NP}-hard even when all values are encoded in unary. Under the widely believed assumption of \( P \neq NP \), such a problem cannot have a pseudo-polynomial solution.

4.3.2 Reduction from Hamiltonian Path Problem

In order to show the hardness we provide a reduction from the classical Hamiltonian Path Problem (or rather its complement).

\textbf{Definition 4.3.1.} \textit{The problem of deciding whether a directed graph} \( G \) \textit{with} \( n \) \textit{vertices contains a simple path with} \( n \) \textit{vertices, i.e.,} \( n \) \textit{unique vertices, is called the Hamiltonian Path Problem.} \( \square \)

\textbf{Proposition 4.3.2 ([29]).} \textit{The Hamiltonian Path Problem is NP-hard in the strong sense.} \( \square \)

We provide a reduction from the Hamiltonian Path Problem as follows. Given a directed graph \( G \), we construct a task set \( \tau \) with the following properties:

1. If \( G \) contains a Hamiltonian Path, \( \tau \) is infeasible.
2. If \( G \) does not contain a Hamiltonian Path, \( \tau \) is feasible.
3. The number of vertices in the tasks of \( \tau \) and all involved values (labels and constraints) is polynomially bounded in the size of \( G \).
4. Given a constant \( c < 1 \), we can construct \( \tau \) such that \( U(\tau) \leq c \).

The third requirement is necessary to establish \textit{coNP}-hardness in the \textit{strong} sense. The fourth requirement is also necessary since we usually restrict ourselves to a class of task sets with a utilization bounded by a constant \( c < 1 \). We want to show that for any choice of \( c \), the problem stays strongly \textit{coNP}-hard.

For presentation reasons, we first assume \( c = 1/2 \). A simple generalization to arbitrary \( c < 1 \) is given afterwards. We construct the task set \( \tau \) from graph \( G \) with \( n \) vertices as follows. The task set contains two tasks:

- The first task \( T_1 \) contains just one vertex \( u_1 \) without any edges, and the label \( \langle e(u_1), d(u_1) \rangle = \langle 1, n \rangle \).
- The second task \( T_2 \) uses \( G \) as underlying graph. All vertices are labeled with \( \langle 1, 1 \rangle \) and all edges with 1. Further, its constraint set contains self-loops with label \( 2n \), i.e., \( C(T_2) := \{ (v, v, 2n) \mid v \in V(T_2) \} \).
Note that all $n$ constraints are potentially overlapping, so this is not a case in which the countdown compression optimization from Section 4.2.3 applies. We illustrate the construction with the following example.

We illustrate the construction with the following example.

(a) Given digraph $G$

(b) Constructed task set $\tau$:

![Figure 4.4](image)

Figure 4.4. Example of constructing task set $\tau = \{T_1, T_2\}$ from a given digraph $G$ containing 6 vertices. This example contains a Hamiltonian Path, making $\tau$ infeasible.

**Example 4.3.3.** Consider graph $G$ in Figure 4.4(a) with 6 vertices. We note that $G$ contains a Hamiltonian Path. Figure 4.4(b) shows the constructed task set. Along the timed path $(v_1, 1, v_3, 1, v_5, 1, v_2, 1, v_6, 1, v_4)$, $T_2$ may release 6 jobs. This path corresponds to the Hamiltonian Path from $G$ and can cause an execution demand of 6 within 6 time units (5 for releasing all jobs, 1 for the last deadline). Together with the job from $T_1$, this task set is clearly infeasible. □

We now check the four properties from above. First, if there is a Hamiltonian Path in $G$, task $T_2$ may release $n$ jobs along that path within $n - 1$ time units. Together with $T_1$, the system is overloaded in an interval of length $n$.

**Lemma 4.3.4.** If graph $G$ contains a Hamiltonian Path, then $\text{dbf}_\tau(n) \geq n + 1$. □

**Proof.** Let $G$ be a digraph with $n$ vertices containing a Hamiltonian Path $\pi = (\pi_0, \ldots, \pi_{n-1})$ and let $\tau = \{T_1, T_2\}$ be the constructed task set. We already have $\text{dbf}_{T_1}(n) = 1$ since the only job that can be released by $T_1$ has an execution demand of 1 and a deadline of $n$.  

61
For $dbf_{\tau_2}(n)$, we consider the timed path $\pi = (\pi_0, 1, \pi_1, \ldots, 1, \pi_{n-1})$ that is constructed from $\pi$ by inserting delays of 1. Since $\pi$ is simple, all vertices are unique in $\pi$, making $\pi$ a legal timed path in $G(T_2)$, because all edge labels are 1 and all additional timing constraints only concern re-visiting of vertices. Now we have with $l = n - 1$:

$$e(\pi) := \sum_{i=0}^{l} e(\pi_i) = n,$$

$$d(\pi) := \sum_{i=0}^{l-1} \delta_i + d(\pi_l) = (n - 1) + 1 = n.$$

Thus, $dbf_{\tau_2}(n) \geq n$ since $\pi$ shows that $T_2$ can create an execution demand of $n$ within a time interval of $n$.

Together, $dbf_{\tau}(n) = dbf_{\tau_1}(n) + dbf_{\tau_2}(n) \geq n + 1$. 

Second, if there is no Hamiltonian Path in $G$, then the constraints from the constructed $C(T_2)$ prevent the system from overloading, since within up to $2n$ time units, $T_2$ can only create a demand of at most $n - 1$.

**Lemma 4.3.5.** If graph $G$ does not contain a Hamiltonian Path, then $\forall t \geq 0 : dbf_{\tau}(t) \leq t$. \hfill \Box

**Proof.** Assume $G$ does not contain a Hamiltonian Path. We want to show $dbf_{\tau}(t) \leq t$ for all $t \geq 0$ and do a case distinction for $t$. Recall that $dbf_{\tau}(t)$ includes only jobs that can be both released and have their deadlines within an interval of size $t$.

$t \in [0, n)$: The job of $T_1$ does not count into such an interval since its deadline is $n$. Further, $T_2$ can only release up to $\lfloor t \rfloor$ jobs within an interval of length $t < n$ which also have their deadlines within the interval. Thus, $dbf_{\tau}(t) \leq \lfloor t \rfloor \leq t$.

$t \in [n, 2n)$: Task $T_1$ can release its job and contributes 1 to $dbf_{\tau_1}$. Further, task $T_2$ can only release up to $n - 1$ jobs within an interval of that size, since vertices cannot be revisited (recall constraints from $C(T_2)$) and there is no simple path that visits all vertices, by assumption. Consequently, $dbf_{\tau}(t) \leq n \leq t$.

$t \geq 2n$: Within every additional $2n$ time units, $T_2$ can only release up to $n - 1$ more jobs. Together with the previous insight, we derive also in this case that $dbf_{\tau}(t) \leq t$. \hfill \Box

Finally, we summarize the hardness result in the following theorem.

**Theorem 4.3.6.** For any constant $c < 1$, the feasibility problem for EDRT task sets $\tau$ with $U(\tau) \leq c$ is coNP-hard in the strong sense. \hfill \Box
Proof. We first assume \( c \in [1/2, 1) \). In that case, the construction introduced above provides a task set \( \tau \) with \( U(\tau) \leq 1/2 \leq c \). The task set’s utilization is at most 1/2, since \( U(T_1) = 0 \) with \( T_1 \) being acyclic, and \( U(T_2) \leq 1/2 \) because any of the \( n \) vertices can only be re-visited after at least \( 2n \) time units. Therefore, even in the presence of a Hamiltonian Path, \( dbf_{T_2}(t) \leq dbf_{T_2}(t - 2n) + n \) for \( t \geq 2n \), resulting in the claimed utilization.

The first two properties from above for a proper reduction are satisfied by Lemmas 4.3.4 and 4.3.5 using the exact \( dbf \) characterization from Theorem 3.1.4. The third property is also clear since all numbers and values are linearly bounded in \( n \). Thus, for \( c \in [1/2, 1) \), we are done.

Finally, in order to satisfy also the last property, note that for \( c < 1/2 \) we can change the task construction by setting \( C(T_2) := \{(v, v, n/c) \mid v \in V(T_2)\} \) instead. This reduces \( U(T_2) \) to at most \( c \) and leaves all other properties satisfied.

We note that the presented reduction constructs a task set with constrained deadlines. (For arbitrary deadlines, only one task is necessary and the construction can be even simpler.) Thus, even when restricting to constrained deadlines, the problem remains strongly \( \text{coNP} \)-hard.

Remark 4.3.7. Theorem 4.3.6 holds even for the restricted class of task sets with constrained deadlines. □

4.4 Tractability for Subclasses

We conclude this chapter with complexity considerations for subclasses of the  
EDRT task model. Specifically, restrictions of the underlaying graph \( G(T) \) 
lead to different complexity results.

Definition 4.4.1. A \( \text{DRT} \) task system \( \tau \) is
- a recurring branching (RB) [8] task system if for each task \( T \in \tau \), a 
subset \( E_{\text{tree}}(T) \) of its edges forms a directed tree \( G_{\text{tree}}(T) = (V(T), E_{\text{tree}}(T)) \) 
and all remaining edges \( E(T) \setminus E_{\text{tree}}(T) \) go from the tree leaves to its 
root. Further, each simple cycle \( \pi \) has the same duration \( p(\pi) \);
- a generalized multiframe (GMF) [12] task system if \( G(T) \) is a cycle 
graph for each task \( T \in \tau \);
- a non-cyclic generalized multiframe (ncGMF) [46] task system if \( G(T) \) 
is a complete graph for each task \( T \in \tau \) such that for each edge \( (u, v) \in V(T) \), the value \( p(u, v) \) is independent of \( v \).

We extend RB, GMF and ncGMF tasks with a set of global timing constraints and denote the extended versions with E-RB, E-GMF and E-ncGMF, respectively. □
4.4.1 Hardness for E-RB and E-ncGMF

For E-RB and E-ncGMF task systems, we can show that the feasibility problem is already coNP-hard in the strong sense. Intuitively, their non-deterministic branching possibilities already provide sufficient complexity to make the problem too hard for pseudo-polynomial algorithms.

**Theorem 4.4.2.** For any constant $c < 1$, the feasibility problem for E-RB and E-ncGMF task sets $\tau$ with $U(\tau) \leq c$ is coNP-hard in the strong sense. □

**Proof.** The hardness proofs for both subclasses are similar to the proof we give above for EDRT by reduction from the Hamiltonian Path Problem. Given a directed graph $G = (E, V)$ with $n$ vertices, we only sketch the constructions of task sets in both models for the case $c = 1/2$. As before, the resulting task set is feasible if and only if the given path contains a Hamiltonian Path.

For both models, we define $T_1$ similar to the above construction for EDRT, i.e., it contains one vertex $u_1$ with label $\langle e(u_1), d(u_1) \rangle = \langle 1, 2n \rangle$. This task is missing its deadline in case there is a Hamiltonian Path in $G$. Further, we construct a task $T_2$ as follows for each task model.

**E-RB Model.** Graph $G(T_2)$ contains vertex $u_2$ as root in the underlaying tree and vertices $v_1, \ldots, v_n$ as its leaves. Recall that there are edges $(u_2, v_i)$ and $(v_i, u_2)$ for each $i$. (We only call the vertices $v_i$ “leaves” because this is their role in the underlaying tree.) All vertices are labeled with $\langle 1, 1 \rangle$ and all edges with 1. Further, we identify all leaves with a corresponding vertex in the given graph $G$. With this implicit mapping, we have the following global inter-release separation constraints for all leaves $u$ and $v$:

$$(u, v, 3) \text{ if } (u, v) \notin E,$$

$$(v, v, 4n) \text{ for all } v \in V.$$

Intuitively, this construction allows a path through $G(T_2)$ to visit $n$ tree leaves with 2 time units in between successive releases of tree leaves if and only if successive visits of tree leaves have a corresponding edge in $G$ (ensured by the first set of constraints) and each tree leaf is only visited once (ensured by the second set of constraints).

**E-ncGMF Model.** The construction of $G(T_2)$ in this case is very similar. We construct a complete graph with $n$ vertices, all vertices are labeled with $\langle 2, 2 \rangle$ and all edges with 2. Again, we create the same set of constraints for all vertices $u$ and $v$ after identifying the vertices in $G(T_2)$ with the vertices of $G$:

$$(u, v, 3) \text{ if } (u, v) \notin E,$$

$$(v, v, 4n) \text{ for all } v \in V.$$
As for the E-RB construction, a path keeping the processor busy for \(2n\) time units must visit each vertex exactly once and essentially follow edges from \(G\). This is possible if and only if \(G\) contains a Hamiltonian Path.

Note that both constructions lead to \(U(\tau) \leq 1/2\) for the resulting task set. For constants \(c < 1/2\), a task set with \(U(\tau) \leq c\) can be constructed by simply scaling the inter-release constraint value from \(4n\) to \(2n/c\) in both cases, similar to the construction above for EDRT task systems.

4.4.2 Pseudo-polynomial Test for E-GMF

The key insight for an efficient analysis procedure for cycle graphs with global constraints, i.e., E-GMF task systems, is that the lack of non-deterministic branching leads to deterministic path behavior which can be exploited as follows. Since there are no branches, there is only one path leading to any particular vertex, allowing a priori knowledge of the minimal time delay since every other vertex has been visited last. In other words, since there is no exponential explosion in the number of paths, they can in fact be efficiently enumerated, leading to the following result.

**Theorem 4.4.3.** For an E-GMF task set \(\tau\) with \(U(\tau) \leq c\) for some constant \(c < 1\), feasibility can be decided in pseudo-polynomial time. □

**Proof (Sketch).** The analysis procedure for DRT tasks from Chapter 3 can be applied, except that we need to change the way in which all demand pairs are generated. Instead of using the demand triple abstraction, we can simulate all paths up to a certain deadline \(D\). Each time we extend a path with the next vertex, all global constraints that involve the new vertex need to be checked which can be done with an efficient look-up. Since for a graph of \(n\) vertices, a path can only start from \(n\) different positions and develops from there in a deterministic way, all possibilities can be enumerated in time polynomial in \(D\). □

**Utilization**

A subtle detail is the computation of the utilization of an E-GMF task set. As an E-GMF task is represented by a cycle graph, finding the most dense cycle may seem trivial, but cyclically overlapping global constraints can lead to the utilization being a non-obvious value. We give an example in Figure 4.5(a).

The approach for \(k\)-EDRT tasks cannot be applied, since we do not assume a bound of the number of global constraints. Thus, translating an E-GMF task into a plain DRT task without global constraints as in Section 4.2.1 in order to find the most dense cycle takes exponential time and may therefore be prohibitively slow. However, we sketch a formulation as a linear optimization...
problem that can be solved in polynomial time [36] and results in the utilization, as follows. A legal timed path which is a simple cycle in $G(T)$ can be written as $\pi = (\pi_0, \delta_0, \ldots, \delta_{n-1}, \pi_n)$ with $\pi_n = \pi_0$. We can use all delays $\delta_i$ as variables in a constraint system for which we create the following linear inequalities.

$$\delta_i \geq p(\pi_i, \pi_{i+1}) \quad \text{for each vertex } \pi_i,$$

$$\delta_i + \ldots + \delta_j \geq \gamma \quad \text{for each constraint } (\pi_i, \pi_j, \gamma) \in C(T).$$

In the second type of constraints, $j$ may need to “wrap around” if $j < i$. Note that these constraints exactly define legal timed paths, cf. Section 4.1.2. The linear optimization problem minimizes $\sum_{i=0}^{n-1} \delta_i$ and can be solved in polynomial time. The solution gives the minimal length of a directly repeatable cycle in $G(T)$, resulting in $U(T) = \sum_{v \in V(T)} e(v) / \sum_i \delta_i$. An example is illustrated in Figure 4.5(b).

![Diagram](image)

**Figure 4.5.** Example of an E-GMF task with non-obvious utilization. The utilization $U(T) = 3/19.5$ is obtained via $\pi = (v_1, 7.5, v_2, 6.5, v_3, 5.5, v_1)$ based on the solution to the given optimization problem.

It can be shown that this is indeed the utilization value. We only give some intuition. First, the value is not too small, i.e., the cycle duration not too large. This can be seen by considering the conversion to a DRT task and taking its most dense cycle. This cycle has been proven to correspond to the utilization of the converted task, and since their demand-bound functions coincide (Lemma 4.2.4 on page 55) to $U(T)$ as well. The delays of this cycle yield a solution to the set of constraints by using the average delay for each edge. Second, the value is not too large, since repeated cycling of this timed path would create a demand-bound function which grows asymptotically faster than the actual utilization, contradicting its definition.
The analysis methods presented in the previous two chapters allow insights about general feasibility of task systems, i.e., whether any scheduler may be able to schedule the workload. They also carry over to schedulability with EDF schedulers because of the optimality of EDF. Another practically important scheduler class are static priority schedulers. One advantage of a static priority order of all tasks in the system is that this property reduces run-time complexity of the scheduler. Another advantage is a certain interference robustness at run-time if certain tasks exhibit behaviors beyond the worst-case bounds that were assumed during the analysis. In an EDF system, this may have an effect on all other tasks, but with static priorities, only tasks of lower priorities can be affected.

The schedulability problem for static priority schedulers is different from the feasibility problem in the sense that concrete scheduler behavior must be taken into account. For feasibility analysis, that was not really necessary: the Feasibility Theorem allowed a test based on identifying worst-case demand in any time interval and this demand is independent of the concrete scheduler. If there is too much demand, no scheduler can succeed in scheduling the workload. On the other hand, the Feasibility Theorem shows that an EDF scheduler can successfully schedule any demand that does not clearly overload the system. However, static priority schedulers are not as flexible and may fail to successfully schedule task systems that are generally feasible. Interference patterns may arise which only lead to deadline misses if combined with certain patterns from other tasks. This makes the schedulability problem for static priorities much more difficult.

This chapter provides proofs for the fundamental hardness of analyzing static priority schedulers. Static priority schedulability is shown to be strongly coNP-hard, which contrasts it to EDF schedulability. The results are presented for subclasses of DRT task systems, showing that already supposedly simplifying special cases are hard to analyze.

5.1 Model Subclasses

The first hardness result restricts DRT tasks to tree release structures. This restriction is a finite version of Recurring Branching Tasks [8], i.e., without back edges. In particular, all job sequences generated by these models are
finite. This shows that the hardness does not come from the recurring nature of DRT tasks which, in general, may contain cycles.

The second hardness result restricts DRT tasks to cycle graphs. This removes the ability to model branches, resulting in the Generalized Multiframe (GMF) [12] task model which we already introduced briefly in the previous chapter. Non-deterministic branches are therefore shown not to be the sole source of hardness either.

**Definition 5.1.1.** A DRT task system $\tau$ is

- a branching task system if $G(T)$ is a directed tree for each task $T \in \tau$,
- a generalized multiframe (GMF) task system if $G(T)$ is a cycle graph for each task $T \in \tau$.

Examples are shown in Figure 5.1. Both models are described in more detail in Chapter 9.

![Figure 5.1. Examples of branching and GMF tasks.](image)

5.2 Hardness for Tree Models

Intuitively, the analysis of branching task systems should be easier than for full DRT task systems, since only finite job sequences can be generated. Further, the number of paths in each task is linear, in contrast to exponential for DRT tasks even if the path length is bounded. However, our result shows that already this case is strongly $\text{coNP}$-hard.
5.2.1 Proof Overview

We use the same type of Karp reduction introduced in Section 4.3.1 on page 59, i.e., polynomial-time many-one reductions. We will provide a reduction from the 3-PARTITION problem.

**Definition 5.2.1.** An instance \( I = (A, s, B) \) of 3-PARTITION consists of

1. a set \( A = \{a_1, \ldots, a_{3m}\} \) of \( 3m \) elements,
2. a size function \( s : A \rightarrow \mathbb{N} \), and
3. a bound \( B \in \mathbb{N} \),

such that \( \sum_{a \in A} s(a) = m \cdot B \) and \( B/4 < s(a) < B/2 \) for all \( a \in A \). An instance \( I \) is a positive instance if \( A \) can be partitioned into \( m \) disjoint sets \( P_1, \ldots, P_m \) such that \( \sum_{a \in P_j} s(a) = B \) for all \( P_j \).

It follows from the definition that in case of a positive instance, all partitions \( P_j \) contain exactly 3 elements. Therefore, we call these \( P_1, \ldots, P_m \) derived from a positive instance also a valid 3-partition.

**Proposition 5.2.2 ([24]).** The 3-PARTITION problem is NP-hard in the strong sense.

We use this problem to show in this section that the static priority schedulability problem is strongly coNP-hard for tree models. The proof has the following structure. Given a 3-PARTITION instance \( I \), we construct a branching task set \( \tau \) with a priority order \( Pr \) in Section 5.2.2. We construct \( \tau \) such that it is unschedulable with any static priority scheduler if and only if \( I \) is a positive instance, i.e., a 3-partition exists. Our proof is divided in three parts.

- First, we show in Section 5.2.3 that a positive instance \( I \) results in an unschedulable \( \tau \).
- Second, we show in Section 5.2.4 that if \( \tau \) is missing a deadline in a synchronous roots sequence (SRS), then \( I \) is a positive instance. In an SRS, all tasks are releasing their roots exactly at the same time.
- Third, we show in Section 5.2.5 that the SRS assumption can be dropped if we add a minor extension to the task set construction. For the extended task set, it will be the case that if \( \tau \) misses a deadline for any job sequence, it can also miss a deadline for a synchronous roots sequence.

Note that the reduction must be polynomial-time and all values in \( \tau \) have to be polynomially bounded in the parameters of \( I \).
### 5.2.2 Task Set Construction

Given an instance \( I = (A, s, B) \) of the 3-PARTITION problem, we construct a task set \( \tau(I) \). The idea is to have one task \( T_i \) per element \( a_i \in A \). Each task has a non-deterministic choice for representing its assignment to one of the \( m \) partitions. The choices differ in delay and execution time which are constructed exactly so that only a valid 3-partition can cause the processor to be constantly busy for a long time. Finally, an additional task with lowest priority is constructed, receiving interference from all other tasks. It has a very short execution time, so it will only miss its deadline if the processor is constantly busy for a sufficiently long time. We will see that only in the case of a valid 3-partition, the processor can be kept busy for that long.

We now give the construction details.

1. We first define a constant \( L \) that is used as a deadline which is sufficiently long such that it cannot be missed.

\[
L := \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B
\]

2. For each \( a_i \in A \) with \( i = 1, \ldots, 3m \) we construct a task \( T_i \) with priority \( Pr(T_i) := i \) as follows. It is a tree with a root vertex \( u^{(i)} \) and \( m \) children \( v_1^{(i)}, \ldots, v_m^{(i)} \) which are also the leaves of the tree. The labels are as follows:

\[
\begin{align*}
  e(u^{(i)}) &= 0, \\
  d(u^{(i)}) &= 0, \\
  e(v_j^{(i)}) &= j \cdot s(a_i), \\
  d(v_j^{(i)}) &= L, \\
  p(u^{(i)}, v_j^{(i)}) &= \sum_{k=1}^{j-1} k \cdot B.
\end{align*}
\]

Note that \( p(u^{(i)}, v_1^{(i)}) = 0 \). We illustrate the construction in Figure 5.2.

The idea is that a branch from dummy vertex \( u^{(i)} \) to vertex \( v_j^{(i)} \) represents partition \( P_j \). Task \( T_i \) chooses (non-deterministically) to branch to one of the leaves \( v_j^{(i)} \), thereby expressing \( a_i \in P_j \). The different delays are constructed so that all tasks choosing partition \( P_j \) will have their job released and executed in a window of size \( j \cdot B \). These \( m \) windows are adjacent and the execution times will guarantee that only the existence of a valid 3-partition for \( I \) can create a situation in which the processor is continuously busy during all windows.

The leaf jobs’ execution times are equal to the element size \( s(a_i) \) but they are scaled by a factor corresponding to the index of the branch. This is the central idea to prevent tasks from choosing the “wrong” branch and thus the “wrong” partition.
3. Further, we construct another task $T_{\text{low}}$ with the lowest priority, i.e., $Pr(T_{\text{low}}) := 3m + 1$. The idea is that $T_{\text{low}}$ is the task missing a deadline if and only if there is a valid 3-partition for $I$. Its graph $G(T_{\text{low}})$ contains only one single vertex $v_{\text{low}}$. The labels are

$$e(v_{\text{low}}) := 1, \quad d(v_{\text{low}}) := \sum_{j=1}^{m} j \cdot B.$$ 

The deadline $d(v_{\text{low}})$ is the sum of all window sizes, so $T_{\text{low}}$ may miss its deadline if all windows are busy.

**Example 5.2.3.** Consider an instance $I = (A, s, B)$ with $A = \{a_1, \ldots, a_9\}$, $B = 42$ and the sizes $s$ given via:

<table>
<thead>
<tr>
<th>$a \in A$</th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>$a_4$</th>
<th>$a_5$</th>
<th>$a_6$</th>
<th>$a_7$</th>
<th>$a_8$</th>
<th>$a_9$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s(a)$</td>
<td>15</td>
<td>16</td>
<td>12</td>
<td>16</td>
<td>14</td>
<td>11</td>
<td>15</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

The resulting task set $\tau(I)$ is shown in Figure 5.3. □

### 5.2.3 Positive Partition Instance

We first show that for a positive instance $I$ of $3$-PARTITION, the lowest priority task $T_{\text{low}}$ of the constructed task set $\tau(I)$ can miss its deadline.

**Lemma 5.2.4.** For a positive instance $I$ of $3$-PARTITION, task set $\tau(I)$ is not schedulable with a static priority scheduler. □
Figure 5.3. Example task set τ(I) constructed from the 3-PARTITION instance I in Example 5.2.3. We have \( L = 378 \) in this example.

Proof. Let \( P_1, \ldots, P_m \) be a valid 3-partition of \( I = (A, s, B) \). For each \( a_i \in A \) with \( i = 1, \ldots, 3m \) we construct a job sequence for the corresponding task \( T_i \). Let \( P_j \) be the partition containing \( a_i \). The idea is that the dummy root job is released at time 0 and \( T_i \) then branches\(^1\) according to partition \( P_j \), releasing its job corresponding to \( v_{ij} \) in the \( j \)-th window. Formally, the constructed job sequence is

\[
\rho_i := \left[ (0, 0, u_{(i)}), \left( r_i, j \cdot s(a_i), v_{ij} \right) \right], \text{ with } r_i := \sum_{k=1}^{i-1} k \cdot B.
\]

\(^1\)Note that this lemma can be proven in a simpler way by just releasing all jobs corresponding to vertices \( v_{im} \) of all tasks at the same time as \( T_{low} \). However, our presented proof also holds for the extended task construction in Section 5.2.5, where the simpler version would not suffice.
Note that \( r_i \) is the start time of the \( j \)-th window. Further, we release \( T_{\text{low}} \) also at time 0, which causes it to have a deadline exactly when the \( m \)-th window ends. The corresponding job sequence is

\[
\rho_{\text{low}} := [0, 1, v_{\text{low}}] .
\]

If these sequences are composed to a job sequence \( \rho \) generated by \( \tau \) and scheduled with a static priority scheduler, task \( T_{\text{low}} \) will miss its deadline. The reason is that for each window \( j \in \{1, \ldots, m\} \), the processor is busy during the whole window as follows.

- Window \( j \) is of size \( j \cdot B \) and starts at time \( \sum_{k=1}^{j-1} k \cdot B \). This starting time is the same time point at which window \( j - 1 \) ends.
- Right when window \( j \) starts, the non-zero jobs of all tasks \( T_i \) with \( a_i \in P_j \) are released. Since this is a 3-partition, these must correspond to three elements \( a, b, c \in A \).
- We know that \( s(a) + s(b) + s(c) = B \) since we chose them from a valid 3-partition.
- The corresponding jobs released at the start of window \( j \) have execution times \( j \cdot s(a) \), \( j \cdot s(b) \) and \( j \cdot s(c) \), respectively.
- In summary, window \( j \) has a workload of \( j \cdot s(a) + j \cdot s(b) + j \cdot s(c) = j \cdot B \) which is exactly the window size.

Thus, the processor is busy from time 0 until the end of the last window \( m \), i.e., time \( \sum_{k=1}^{m} k \cdot B \), which is the deadline of \( v_{\text{low}} \). Therefore, \( T_{\text{low}} \) is never executed before its deadline and will inevitably miss it.

The deadline miss scenario in the above proof can be illustrated with the following example.

**Example 5.2.5.** Consider the 3-PARTITION instance \( I \) from Example 5.2.3 on page 71. It is clearly a positive instance, since the following is a valid 3-partition:

\[
P_1 = \{a_1, a_2, a_6\}, P_2 = \{a_3, a_4, a_5\}, P_3 = \{a_7, a_8, a_9\} .
\]

We illustrate the resulting schedule described in the above proof in Figure 5.4. Note that the job execution times exactly fit the three windows. Thus, the processor is busy between time 0 and 252 executing jobs from \( T_1, \ldots, T_9 \) with higher priorities than \( T_{\text{low}} \). Since \( T_{\text{low}} \) has its deadline at time 252, it misses its deadline.

\( \square \)
Figure 5.4. Schedule resulting from the valid 3-partition in Example 5.2.5. The arrows representing job releases are annotated with the elements \( a \in A \) to which the released jobs correspond, in addition to the job of \( T_{low} \). Different colors represent task releases in different windows, i.e., assignment to partitions. Note that \( T_{low} \) has an execution time of just 1, so it is very slim in the figure.

5.2.4 Deadline Miss with SRS

We focus now on the other direction: the construction of a valid 3-partition from a job sequence \( \rho \) in which a task in \( \tau(I) \) misses its deadline. The idea is that schedules which do not correspond to a valid 3-partition cannot cause the processor to be busy in all windows. We illustrate this with the following example.

Example 5.2.6. We use instance I from Example 5.2.3. Consider the following 3-partition which is not valid:

\[
P'_1 = \{a_1, a_2, a_4\}, P'_2 = \{a_6, a_8, a_9\}, P'_3 = \{a_3, a_5, a_7\}.
\]

A schedule according to the construction above in the proof of Lemma 5.2.4 will make all jobs meet their deadlines, since the second window is not entirely busy. We illustrate the schedule in Figure 5.5.

In the remainder of this section, we show that if a schedule does not correspond to a valid 3-partition, some window is not entirely busy and \( T_{low} \) can execute\(^2\). Thus, \( T_{low} \) will always meet all deadlines for negative 3-PARTITION instances.

We first establish the proof under the assumption that \( \rho \) is a synchronous roots sequence.

Definition 5.2.7. For a branching task set \( \tau \), a job sequence \( \rho \) is a synchronous roots sequence (SRS) if for each task \( T \in \tau \), the subsequence \( \rho_T \) generated by task \( T \) satisfies the following:

\(^2\)Generally, a schedule does not need to conform to the construction in the proof of Lemma 5.2.4. That is, tasks may start execution anywhere in the tree at any time. However, it will be established in Section 5.2.5 that it is sufficient to only consider such schedules where the execution starts synchronously at the root vertices.
Figure 5.5. Schedule resulting from the invalid 3-partition in Example 5.2.6. In the end of the second window, the processor does not execute a job of higher priority than \( T_{low} \), so \( T_{low} \) can execute and thus meets its deadline.

1. The first job in \( \rho_T \) corresponds to the root vertex of \( G(T) \) and is released at time 0.
2. All later jobs in \( \rho_T \) are released as early as possible.

This assumption is necessary: otherwise, all tasks \( T_i \) of \( \tau(I) \) could release their job associated with vertex \( v_m^{(j)} \) at the same time as \( T_{low} \) is released. This would clearly cause a deadline miss of \( T_{low} \), no matter whether \( I \) is a positive or a negative instance. Assuming an SRS is the key to prevent such a situation. (However, the SRS assumption is removed in Section 5.2.5 by extending the task set construction.)

Lemma 5.2.8. If there is a synchronous roots sequence \( \rho \) of \( \tau(I) \) in which some job misses its deadline with a static priority scheduler, then \( I \) is a positive instance of 3-PARTITION.

Proof. First we note that it must be the job of \( T_{low} \) which is missing its deadline. The deadlines of all other jobs are either 0 (root dummy vertices with 0 execution time) or \( L \) (leaf vertices), which is larger than all possibly interfering workload of higher priority tasks. Further, without loss of generality, we may assume that all jobs in \( \rho \) are taking the maximum execution time allowed by their releasing vertices. This assumption is safe since increasing the interfering workload will not prevent a deadline miss that is otherwise happening.

The job of \( T_{low} \) that is missing its deadline is released at time 0 and has its deadline at time \( D := \sum_{k=1}^{m} k \cdot B \). Since it is missing the deadline, the processor is busy executing higher priority jobs for strictly more than \( D - 1 \) time units. For presentation reasons we obtain a job sequence \( \rho' \) by removing the job of \( T_{low} \) from \( \rho \). For \( \rho' \) we know that the processor is idle during \([0,D]\) for strictly less than one time unit. We now use the window concept from above for dividing the time interval \([0,D]\) into \( m \) non-overlapping windows. The \( j \)-th window is of size \( j \cdot B \) and starts at time \( \sum_{k=1}^{j-1} k \cdot B \). These time points are exactly the time points in \( \rho' \) at which all tasks \( T_i \) release their non-dummy jobs (because of the SRS assumption). We will now show that during all windows,
\( \rho' \) keeps the processor constantly busy. Based on this we will construct a valid 3-partition by observing which branch each task \( T_i \) chose in \( \rho' \).

We first look at the first window. It starts at time 0 and ends at time \( B \). Jobs are only released at time 0 and then not again before \( B \). This is guaranteed by the inter-release separation constraints on the tree edges and the SRS assumption. We further know that with job sequence \( \rho' \), the processor is idle for strictly less than one time unit during the first window. All execution times are integers, so the processor is in fact not idle at all, or in other words, busy executing jobs throughout the whole window. In particular, there is no idle time at the end of the window. Let \( \sigma_1 \) denote the sum of all \( s(a_i) \) for which task \( T_i \) chose the first branch in \( \rho' \). Since the processor is busy throughout the whole window (and possibly even afterwards) executing jobs released at time 0, we have \( \sigma_1 \geq B \).

The second window has a size of \( 2B \), starting at time \( B \) and ending at time \( B + 2B = 3B \). As in the first window, job releases can only take place at the start of the window and then not again until its end. Since the processor is also busy during this whole window by the same reasoning as above, we again know that there is no idle time at the end of the window. We let \( \sigma_2 \) denote the sum of all \( s(a_i) \) for tasks \( T_i \) that chose the second branch in \( \rho' \). The sum of these jobs’ execution times is thus \( 2\sigma_2 \) since the execution times of \( v_2 \) in all trees are scaled by a factor of 2. Taking into consideration that there may be some workload reaching from the first window into the second window, we can express our knowledge that there is no idle time at the end of the second window as \( \sigma_1 + 2\sigma_2 \geq B + 2B \).

We now generalize this to the \( k \)-th window. For every \( j = 1, \ldots, m \), let \( \sigma_j \) denote the sum of all \( s(a_i) \) for which task \( T_i \) chose the \( j \)-th branch in \( \rho' \). The sum of the jobs’ execution times in this branch is \( j \cdot \sigma_j \) because of the scaling factor during task construction. Similar to the above reasoning we know that there is no idle time at the end of the \( k \)-th window, which we can express as

\[
\sum_{j=1}^{k} j \cdot \sigma_j \geq \sum_{j=1}^{k} j \cdot B. \tag{5.A}
\]

Additionally, we know that the sum of all \( s(a_i) \) is \( m \cdot B \), so the sum of all \( \sigma_j \) is \( m \cdot B \) as well:

\[
\sum_{j=1}^{m} \sigma_j = m \cdot B. \tag{5.B}
\]

From both conditions we derive that \( \sigma_j = B \) for all \( j \), shown in Lemma 5.2.9 below. This means that for each branch \( j \), the sum of the \( s(a_i) \) corresponding to those tasks \( T_i \) that chose this branch \( j \) in \( \rho' \) is exactly \( B \). In other words, for each \( j \), the set of all elements \( a_i \) contributing to \( \sigma_j \) is the set \( P_j \) of a valid 3-partition. \( \blacksquare \)
Lemma 5.2.9. Conditions (5.A) and (5.B) imply $\sigma_j = B$ for all $j = 1, \ldots, m$. □

Proof. For presentation, we re-state both conditions: We apply some arithmetic transformations on both conditions in order to show our goal $\forall j : \sigma_j = B$. First, we set $\hat{\sigma}_j := \sigma_j - B$. Intuitively, $\hat{\sigma}_j$ expresses how “far off” the sum of the supposed partition $P_j$ is from the goal $B$. Our proof goal now becomes $\forall j : \hat{\sigma}_j = 0$ and the conditions become:

\[
\forall k : \sum_{j=1}^{k} j \cdot \hat{\sigma}_j \geq 0, \quad \text{and} \quad \sum_{j=1}^{m} \hat{\sigma}_j = 0. \quad (5.A') \quad (5.B')
\]

Second, we set $\alpha_k := \sum_{j=1}^{k} j \cdot \hat{\sigma}_j$ which is the LHS of Condition (5.A’) for all $k = 1, \ldots, m$. Intuitively, it indicates for each $k$ the idle time until the end of the $k$-th window. (More precisely, negative values are accumulated idle time, positive values are workload that reaches into the following window.) For substitution purposes, we want to express all $\hat{\sigma}_j$ using the new symbols:

\[
\alpha_k = \sum_{j=1}^{k} j \cdot \hat{\sigma}_j \implies \alpha_k - \alpha_{k-1} = k \cdot \hat{\sigma}_k \implies \hat{\sigma}_k = \frac{1}{k} (\alpha_k - \alpha_{k-1}).
\]

We do the substitution in both Conditions (5.A’) and (5.B’) and get:

\[
\forall k : \alpha_k \geq 0 \quad (5.A'') \quad \sum_{j=1}^{m} \frac{1}{j} (\alpha_j - \alpha_{j-1}) = 0 \quad (5.B'')
\]

With straightforward arithmetics, we can derive from Condition (5.B’’):

\[
\sum_{j=1}^{m} \frac{1}{j} (\alpha_j - \alpha_{j-1}) = \sum_{j=1}^{m} \frac{1}{j} \alpha_j - \sum_{j=1}^{m} \frac{1}{j} \alpha_{j-1}
\]

\[
= \sum_{j=1}^{m} \frac{1}{j} \alpha_j - \sum_{j=0}^{m-1} \frac{1}{j+1} \alpha_j
\]

\[
= \frac{\alpha_m}{m} + \sum_{j=1}^{m-1} \alpha_j \left( \frac{1}{j} - \frac{1}{j+1} \right) - \frac{\alpha_0}{1} \left( \text{term is 0} \right)
\]

\[
= \frac{\alpha_m}{m} + \sum_{j=1}^{m-1} \frac{\alpha_j}{j(j+1)} \quad (5.B'') \quad = 0
\]

77
Since the coefficients of all $\alpha_j$ are positive, a non-negative solution to this condition can only exist with $\alpha_j = 0$ for all $j$. Note that Condition (5.A") tells us that only non-negative solutions are allowed.

In conclusion, we have $\forall j : \alpha_j = 0$, which is equivalent to $\forall j : \sigma_j = B$. ■

5.2.5 Deadline Miss without SRS

In order to show the result without the assumption of a synchronous roots sequence, we need to adjust the construction of task set $\tau(I)$. Recall that this assumption was necessary to prevent the tasks from just directly releasing jobs associated with $v^{(i)}_m$ causing high workload. As a solution to this issue, we add a very long job to the root vertex of all $G(T_i)$. We call this job the heavy job of task $T_i$, in contrast to the leaf jobs of $T_i$. As we will see, the introduction of this heavy job guarantees that the worst case interference for $T_{low}$ is created when all $T_i$ synchronously release the jobs associated with their root vertices (which are now their heavy jobs). This consequently results in the SRS from above.

The new task set $\tau'(I)$ differs from the original $\tau(I)$ only in the root vertices of the release trees and adjusted delays and deadlines. The details are as follows.

1. We again use the large constant $L$ defined above. It is sufficiently large to represent a time span which is just as long as the workload that all leaves $v^{(i)}_m$ can create if they are released together.

$$L = \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B$$

2. For each $a_i \in A$ we construct a task $T_i$ with priority $Pr(T_i) := i$ as in Section 5.2.2, but with the following labels:

$$e(u^{(i)}) := L, \quad d(u^{(i)}) := 3mL,$$
$$e(v^{(i)}_j) := j \cdot s(a_i), \quad d(v^{(i)}_j) := 3mL,$$
$$p(u^{(i)}, v^{(i)}_j) := 3mL + \sum_{k=1}^{j-1} k \cdot B.$$

We illustrate the construction in Figure 5.6. In contrast to the task set $\tau(I)$ from Section 5.2.2 above, the new tasks $T_1, \ldots, T_{3m}$ of $\tau'(I)$ are constructed such that a worst-case interference of $T_{low}$ (described below) must contain a heavy job of each $T_j$, followed by a leaf job. We will see that this allows us to reason that all these heavy jobs can be assumed to be released at the same time as $T_{low}$ is released.
Figure 5.6. Illustration of new task construction for task \( T_i \). Note the heavy job released at root vertex \( u^{(i)} \).

3. Again, we further construct another task \( T_{\text{low}} \) with the lowest priority, i.e., \( Pr(T_{\text{low}}) := 3m + 1 \). As before, \( T_{\text{low}} \) is the task missing a deadline if and only if there is a valid 3-partition for \( I \). Its graph \( G(T_{\text{low}}) \) contains again only single vertex \( v_{\text{low}} \) with labels

\[
e(v_{\text{low}}) := 1, \quad d(v_{\text{low}}) := 3mL + \sum_{j=1}^{m} j \cdot B.
\]

Note that \( d(v_{\text{low}}) \) now includes time for the heavy jobs of all \( 3m \) tasks \( T_i \) in addition to the sum of all window sizes. Thus, if the processor is first busy executing a heavy job of each task \( T_i \) and then leaf jobs of all \( T_i \) during all windows, \( T_{\text{low}} \) is missing its deadline.

An example execution is illustrated in Figure 5.7, based on task set \( \tau'(I) \) resulting from instance \( I \) in Example 5.2.3 on page 71. All heavy jobs are released at time 0, keeping the processor busy for \( 3mL \) time units. After that, the leaf jobs are released according to their partitions, just as before in Section 5.2.4. This keeps the processor constantly busy and causes a deadline miss for \( T_{\text{low}} \). Conversely, if there is a deadline miss, it again must be the job of \( T_{\text{low}} \) missing the deadline, since all other jobs have deadlines \( 3mL \) which is again more than all possible interfering workload of higher priority tasks.

Based on these observations, it is easily verified that both Lemmas 5.2.4 and 5.2.8 also hold for the new task set \( \tau'(I) \), i.e., \( I \) is a positive instance if and only if \( \tau'(I) \) is unschedulable in SRS. We will now focus on removing the SRS assumption. That is, we will show that the synchronous roots sequence assumed in Lemma 5.2.8 constitutes the worst case interference for task \( T_{\text{low}} \). In other words, we will show that any job sequence with a deadline miss for \( T_{\text{low}} \) can be transformed into one with an SRS which still causes a deadline miss.
Lemma 5.2.10. If there is a job sequence generated by $\tau'(I)$ such that $T_{\text{low}}$ misses a deadline when scheduled with a static priority scheduler, then there is a synchronous roots sequence in which $T_{\text{low}}$ misses a deadline as well.

Proof. By shifting all release times and deadlines in a given job sequence $\rho$, we may assume that the job of $T_{\text{low}}$ which is missing its deadline is released at time 0 and has its deadline at time $D = 3mL + \sum_{j=1}^{m} j \cdot B$. We now transform $\rho$ such that the deadline miss of $T_{\text{low}}$ occurs in a synchronous roots sequence. During all stages of the transformation, we ensure that the processor work during the time interval $[0, D]$ does not decrease.

1. The first change we apply in case the processor is busy at time 0, i.e., right when $T_{\text{low}}$ has its release time. We find a time point $t < 0$ which is the earliest time point such that the processor is continuously busy executing jobs until time 0. Thus, there is a busy period of length $\delta := 0 - t$ right before time point 0. We now change $\rho$ by shifting all release times and deadlines of tasks $T_{1}, \ldots, T_{3m}$ by $\delta$. This change ensures that
   
   • by construction, the processor is now idle right when $T_{\text{low}}$ is released at time 0, and
   • during $[0, D]$, the interference of $T_{\text{low}}$ by higher priority tasks does not decrease. This is because we moved $\delta$ time units of interfering work into the interval, but at most $\delta$ out of the interval.

2. Because of the above step, we may assume that the processor was actually idle right when $T_{\text{low}}$ was released at time 0. For each $T_{i}$ that does not have the release of a heavy job at time 0, we can therefore do the following:
   
   • If $T_{i}$ releases a heavy job at a time $t'$ inside the interval $(0, D]$, we shift all jobs of $T_{i}$ in $\rho$ by $-t'$, i.e., to earlier time points. Since $t'$ is inside the interval, this does not move the execution of any job of $T_{i}$ out of the interval, but only potentially moves work into the
interval. Thus, the total work inside the interval does not decrease, and we end up with $T_i$’s heavy job being released at time 0.

- Otherwise, if $T_i$ does not release a heavy job inside the interval but a leaf job instead, we move it together with its preceding heavy job (already finished before time 0) so that the heavy job is released at time 0. If there is no such heavy job release, we just insert one at the latest time possible before doing the described transformation. In this step we are moving $L$ time units of work into the interval (that is, the heavy job of $T_i$), but less than $L$ out of the interval (at most a leaf job). Again, work inside $[0,D]$ does not decrease.

- Finally, if $T_i$ does not release any job within $[0,D]$ we can just delete all jobs of $T_i$ from $\rho$ and add its heavy job at time 0. The deletion did not remove any work from the interval by assumption, so in summary the amount of work inside $[0,D]$ increases.

3. After the above steps, there is no job in $\rho$ which is released before $[0,D]$. Further, we can remove all jobs from $\rho$ which are released at or later than $D$ since they do not influence the work inside $[0,D]$. Finally, releasing the leaf jobs inside the window $[0,D]$ as early as possible will also not decrease the amount of work inside $[0,D]$.

In summary, the construction transforms $\rho$ into a synchronous roots sequence. Since the work of tasks with priority higher than $T_{low}$ in the interval $[0,D]$ did not decrease, $T_{low}$ still misses its deadline. ■

We conclude Section 5.2 with the main theorem for branching task models, i.e., with tree release structure.

**Theorem 5.2.11.** For branching task models, the schedulability problem for static priority schedulers is $coNP$-hard in the strong sense. 

**Proof.** Given an instance $I$ of $3$-PARTITION we described a reduction to an instance $\tau'(I)$ of the static priority scheduling problem. We established that

1. a positive instance $I$ leads to a negative instance $\tau'(I)$, Lemma 5.2.4, and
2. a negative instance $I$ leads to a positive instance $\tau'(I)$, Lemmas 5.2.8 and 5.2.10.

Since $3$-PARTITION is $NP$-hard, the static priority scheduling problem for branching task models is $coNP$-hard. Further, this reduction can clearly be executed in polynomial time, and all values in $\tau'(I)$ are bounded polynomially in the values of $I$. With $3$-PARTITION being $NP$-hard in the strong sense, our problem is shown $coNP$-hard in the strong sense. ■
5.3 Hardness for Cycle Models

In this section, we show that the static priority scheduling problem is strongly coNP-hard also for GMF models, cf. Definition 5.1.1 on page 68. The central idea here is that we can construct a GMF task set that exhibits essentially the same behavior as the branching task in Section 5.2 above.

For branching task models, the idea behind the task construction was to have one task $T_i$ for each of the $3m$ elements $a_i \in A$. Each task includes a branch for choosing one of the $m$ partitions $P_j$ which $a_i$ should be part of. For GMF models, we would like to have a similar construction, even though GMF tasks do not allow branches. However, we can unroll the tree construction from the previous section such that all edges are now serially included in a cycle graph. The non-deterministic choice of the branch in the original branching task now translates into a non-deterministic choice about at which vertex to start in the cycle graph of the constructed GMF task.

Formally, we construct a GMF task set $\tau''(I)$ from a 3-PARTITION instance $I$ as follows.

1. Just as in Section 5.2, we use a large constant.

$$L = \sum_{i=1}^{3m} m \cdot s(a_i) = m^2 B$$

2. For each $a_i \in A$ we construct a GMF task $T_i$ with priority $Pr(T_i) := i$ as follows. It includes $m$ vertices $u_1, \ldots, u_m$ representing $m$ copies of the root vertex in the construction of each $T_i \in \tau'(I)$ from Section 5.2. Further, it also includes $m$ vertices $v_1, \ldots, v_m$ corresponding to the leaf vertices in $\tau'(I)$. The edges of $G(T_i)$ alternate between vertices $u_j$ and $v_j$, i.e., for all $j = 1, \ldots, m$, we have

- an edge $(u_j, v_j)$ corresponding to a tree edge in $\tau'(I)$, and
- an edge $(v_j, u_{j+1})$ connecting to the next root vertex copy. (We use $u_{m+1} := u_1$ here to simplify notation.) The label of this edge needs to be sufficiently large so that $T_i$ cannot release a heavy job too soon after a leaf job, which could create too high workload in case of a negative instance $I$.

The labels are similar to those in $\tau'(I)$:

$$e(u_j) := L, \quad d(u_j) := 3mL,$$

$$e(v_j) := j \cdot s(a_i), \quad d(v_j) := 6mL,$$

$$p(u_j, v_j) := 3mL + \sum_{k=1}^{j-1} k \cdot B, \quad p(v_j, u_{j+1}) := 6mL.$$
Figure 5.8. Illustration of constructing GMF task $T_i$. Edges corresponding to tree edges in the tree construction are solid. The dashed edges are connecting to the next root vertex copy.

We illustrate the construction in Figure 5.8. Note that the label of all edges $(v_j, u_{j+1})$ is $6mL$ which is twice the work that all heavy jobs can create together. This guarantees that at the release of the next heavy job, the processor is not executing work that was connected to the previous leaf job. This construction allows us to apply the reasoning from Section 5.2.

3. Just as before, we construct another task $T_{low}$ with the lowest priority $Pr(T_{low}) := 3m + 1$ and only a single vertex $v_{low}$ with a self loop. The labels are the same as in Section 5.2 for $\tau'(I)$, in addition to the long waiting delay from above.

$$e(v_{low}) := 1, \quad d(v_{low}) := 3mL + \sum_{j=1}^{m} j \cdot B, \quad p(v_{low}, v_{low}) := 6mL$$

A synchronous roots sequence for $\tau'(I)$ translates into a job sequence for $\tau''(I)$ which releases a job from a vertex $u_j$ at time 0 for each task $T_i$ in $\tau''(I)$ where each task releases only two jobs. Conversely, because of the long $6mL$ delays, any job sequence generated by $\tau''(I)$ corresponds piecewise to a sequence that is generated by $\tau'(I)$. With both insights, it is easily verified that the three Lemmas 5.2.4, 5.2.8 and 5.2.10 also hold for $\tau''(I)$. Consequently, the main theorem also holds for GMF models.

**Theorem 5.3.1.** For GMF models, the schedulability problem for static priority schedulers is coNP-hard in the strong sense. \hfill □

**Proof.** By the above discussion, similar to Theorem 5.2.11. □
5.3.1 Special Case of Multiframe Tasks

We conclude this chapter showing that the result holds already for a special case of GMF task systems where frames only differ in execution time bounds, called Multiframe (MF) tasks [35]. This establishes a rather precise borderline of tractability for SP scheduling beyond periodic tasks for which SP scheduling can be solved in pseudo-polynomial time by classical response-time analysis.

**Definition 5.3.2.** A GMF task system $\tau$ is a multiframe (MF) task system if for all tasks $T$, deadlines are implicit and the edge labels of its graph $G(T)$ have the same value for all edges of $G(T)$. □

The proof for this subclass of GMF tasks is based on the observation that the constructed GMF task set causing interference can be converted into an MF task set by splitting the frames into pieces of equal duration.

**Lemma 5.3.3.** For $D \in \mathbb{N}$ and a GMF task set $\tau$ with $n$ tasks, an MF task set $\tau'$ can be constructed with the following properties.

1. Task set $\tau'$ can be constructed from $\tau$ in pseudo-polynomial time. Further, all vertex and edge labels in $\tau'$ are polynomially bounded in the labels of $\tau$.

2. $\tau'$ can keep a maximal interval of $n \cdot D$ time units completely busy if and only if $\tau$ can keep a maximal interval of $D$ time units completely busy.

3. If $\tau$ is SP schedulable, then $\tau'$ is SP schedulable. □

**Proof.** We replace each vertex $v$ and its outgoing edge $(v,v')$ with a chain of vertices $v_1,\ldots,v_{p(v,v')}$. All new edges, i.e., $(v_i,v_{i+1})$ and the new edge $(v_{p(v,v')},v')$ are labeled with $n$. Further, each vertex is labeled with:

$$\langle e(v_i),d(v_i) \rangle := \begin{cases} 
\langle 1,n \rangle & \text{if } i \leq e(v), \\
\langle 0,n \rangle & \text{otherwise}.
\end{cases}$$

The first and third property of the lemma are easily verified for the constructed task set which is clearly an MF task set. Note that each frame uses only 1 time unit of execution time within its duration of $n$ time units, leading to guaranteed schedulability of $n$ tasks.

However, we still need to guaranteed the second property. If task set $\tau$ can create a busy period of $D$ time units, our constructed task set so far only creates $D$ time units of execution demand within $n \cdot D$ time units. In order to compensate for this, we create an extra task $T_{n+1}$ with lowest priority which is periodic and has an execution time of $(n-1) \cdot D$ and period $n \cdot D$. The other two properties still hold. □
Note that this lemma does preserve SP schedulability but not necessarily SP unschedulability, i.e., if the given GMF task set is unschedulable, the result could be schedulable, which means that the construction cannot be used as a general reduction of SP schedulability for GMF task sets to MF task sets. However, this property is sufficient for our purposes.

Using Lemma 5.3.3, the result follows: the constructed task set $\tau''(I)$ in Section 5.3 is converted into an MF task set by applying the lemma to all tasks except the one with lowest priority. Replacing $T_{low}$ with a new task where the deadline is scaled by $n$, the number of tasks of higher priority, completes the conversion. The construction guarantees that the task set only grows pseudo-polynomially and also the label values are still polynomially bounded. Thus, the hardness carries over to the restricted class.

**Corollary 5.3.4.** For MF models, the schedulability problem for static priority schedulers is coNP-hard in the strong sense.
6. Combinatorial Abstraction Refinement

The previous chapters have demonstrated that there is a difference in complexity of the DRT schedulability analysis problems for EDF and static priority schedulers. While a pseudo-polynomial time analysis method for EDF schedulability was described in Chapter 3, the schedulability problem for SP schedulers was shown to be strongly coNP-hard in Chapter 5. However, static priority schedulers are of great practical importance. In this chapter, we develop an analysis method which solves this problem rather efficiently for typical problem instances. While it still runs in exponential time in the worst case, a prototype implementation shows run-times which are comparable to the pseudo-polynomial time EDF schedulability test from Chapter 3 on randomly generated task sets (cf. Chapter 8).

We call the central approach for our method Combinatorial Abstraction Refinement since it approaches the combinatorial problem by iteratively refining a given abstraction until a precise result is obtained. The method is a very general one which can be potentially applied to other combinatorial problems that contain certain abstraction structures. We outline these in the end of this Chapter.

6.1 Overview

In this section, we give an overview of our algorithm for checking SP schedulability and SP feasibility. As it turns out, both problems are very similar in that they can be reduced to the same fundamental test called lowest-priority feasibility of single tasks.

6.1.1 Lowest-Priority Feasibility

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

**Definition 6.1.1.** For a task set \( \tau = \{T_1, \ldots, T_N\} \), a task \( T \in \tau \) is lowest-priority feasible in \( \tau \) if there is a priority order \( Pr \) with \( Pr(T) = N \) such that \( T \) does not miss any deadlines if \( \tau \) is SP scheduled with \( Pr \). \( \Box \)

Note that this definition does not state anything about deadline misses of any other tasks in \( \tau \). Further, we will now see that if \( T \in \tau \) is lowest-priority
feasible in $\tau$, then this property is independent of the relative priorities of all other tasks in $\tau$.

**Lemma 6.1.2.** 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 deadlines 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 $T$. Further, in permutations $Pr'$ with $Pr'(T) < N$, task $T$ experiences less interference from tasks of higher priority and is therefore also schedulable. ■

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

### 6.1.2 SP Schedulability

Given a task set $\tau = \{T_1, \ldots, T_N\}$ with a priority order $Pr$, SP schedulability of $\tau$ 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 6.1.3.** 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 $\tau_{\leq T}$ defined as:

$$\tau_{\leq T} := \{ T' \mid Pr(T') \leq Pr(T) \}. \quad \square$$

**Proof.** By above discussion. ■

### 6.1.3 SP Feasibility: Audsley’s Algorithm

Checking SP feasibility of a task set $\tau$ is possible using a similar method which is usually called Audsley’s Algorithm [4]. First, check all $T \in \tau$ for lowest-priority feasibility in $\tau$. 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, then $\tau$ is not SP feasible. This method has the additional advantage of synthesizing a priority order for SP feasible task sets by taking the reverse order in which the tasks are found to be lowest-priority feasible.
Lemma 6.1.4. 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 \( \tau \) and \( \tau \setminus \{T\} \) is SP feasible.

Proof. It is clear that if this test succeeds, \( \tau \) is indeed SP feasible since the synthesized priority order automatically satisfies the condition in Lemma 6.1.3 from above.

Conversely, let the test fail for some subset \( \tau' \subseteq \tau \) but assume there is a priority order \( Pr \) for which \( \tau \) is SP schedulable. Of all tasks in \( \tau' \), some task \( T \in \tau' \) is assigned lowest priority by \( Pr \). Since \( \tau \) 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 \( \tau \). However, \( \tau' \) is just a subset of \( \tau \), therefore the interference experienced by \( T \) from all other tasks in \( \tau' \) cannot be larger (Lemma 6.1.2). Thus, \( T \) must be lowest-priority feasible in \( \tau' \), contradicting the assumption that the test failed for \( \tau' \).

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.

6.1.4 Vertex Testing

It is sufficient to test all vertices of a task separately in order to conclude that the task is lowest-priority feasible. In fact, only the scheduling window for jobs corresponding to each vertex needs to be considered. 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 \( \tau_{\text{high}} \) 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 \( \tau_{\text{high}} \) or just schedulable if \( \tau_{\text{high}} \) is clear from the context. Our algorithm for testing schedulability of a single vertex is described in Section 6.2.

Lemma 6.1.5. Given a task set \( \tau \), a task \( T \in \tau \) is lowest-priority feasible if and only if all vertices \( v \in V(T) \) are schedulable with interference set \( \tau \setminus \{T\} \).

Proof. By above discussion.

\(^1\)Another important condition for this is that all jobs released by the same task do have the same priority, i.e., this method is not directly applicable to a scheduler where different vertices could be assigned different static priorities.
Critical Vertices

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

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

Assume that \( v_1 \) is schedulable with some interference set \( \tau \). 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 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 the 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 only \( O(\sqrt{n}) \), dramatically reducing the number of iterations of a loop that tests all vertices individually for schedulability. Further, as we will see in Section 6.2, 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, i.e., a subset, thus causing less interference. We summarize this concept as follows.

**Definition 6.1.6.** For a task set \( \tau \) with priority order \( Pr \) and tasks \( T, T' \in \tau \), we say that \( v \in V(T) \) dominates \( v' \in V(T') \), written \( v \succeq v' \), if and only if:

1. \( e(v) \geq e(v') \),
2. \( d(v) \leq d(v') \) and
3. \( Pr(T) \geq Pr(T') \).

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 \( v' \succeq v \) 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.

6.2 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 $(e, d)$, i.e., with WCET $e$ and deadline $d$. We want to check whether a given task set $\tau$ 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 $\tau$ do not matter in this case.

A naïve approach for this test could be as follows. For each task $T \in \tau$, pick a path $\pi^{(T)}$. Given the set of paths $\{\pi^{(T)}\}_{T \in \tau}$, the synchronous arrival sequence (SAS) 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 6.1 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.

![Figure 6.1](image)

Figure 6.1. Example of simulating a synchronous arrival sequence in a time interval of size 30. The interference set is $\tau = \{T, T'\}$ with $T$ from Figure 6.2 on page 92 and $T'$ a periodic task with $(e, d, p) = (2, 8, 8)$. From task $T$, we simulate path $(v_4, v_1, v_2)$. In this concrete scenario, 9 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 6.3 introduces a method for reducing the number of combination tests.
6.2.1 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 $\pi$ with a request function which for each $t$ returns the accumulated execution requirement of all jobs that $\pi$ may release during the first $t$ time units.

**Definition 6.2.1 (Request Function)**. For a path $\pi = (v_0, \ldots, v_l)$ through the graph $G(T)$ of a task $T$, we define its request function as

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

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 $\pi$ have request functions that are identical up to the duration $p(\pi)$ of that prefix. We give an example in Figure 6.3.

![Figure 6.2. Example DRT task used in this chapter.](image)

**Figure 6.2. Example DRT task used in this chapter.**

![Figure 6.3. Example of request functions on $[0, 30]$. Paths are taken from $G(T)$ in Figure 6.2 with $\pi = (v_4, v_1, v_2)$, $\pi' = (v_5, v_4, v_1)$ and $\pi'' = (v_2, v_1, v_2)$.](image)

**Figure 6.3. Example of request functions on $[0, 30]$. Paths are taken from $G(T)$ in Figure 6.2 with $\pi = (v_4, v_1, v_2)$, $\pi' = (v_5, v_4, v_1)$ and $\pi'' = (v_2, v_1, v_2)$.**
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 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 $\vec{\pi} = (\pi(T_1), \ldots, \pi(T_N))$ denote an element of $\Pi(\tau)$, i.e., a single combination of paths.

**Theorem 6.2.2.** A vertex $v$ is schedulable with interference set $\tau$ if and only if

$$\forall \vec{\pi} \in \Pi(\tau) : \exists t \leq d(v) : e(v) + \sum_{T \in \tau} rf_{\pi(T)}(t) \leq t.$$  \hspace{1cm} (6.1)

**Proof.** Assume Condition (6.1) holds but $v$ is unschedulable. Because of the latter, there must be a combination of paths $\vec{\pi} = (\pi(T_1), \ldots, \pi(T_N))$ 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 $t \leq d(v)$, tasks from $\tau$ 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

$$\sum_{T \in \tau} rf_{\pi(T)}(t) > t - e(v)$$  \hspace{1cm} (6.2)

for each $t \leq d(v)$. This contradicts the assumption that Condition (6.1) holds.

Assume now that $v$ is schedulable but Condition (6.1) does not hold. Because of the latter, there is $\vec{\pi} \in \Pi(\tau)$ such that Condition (6.2) holds for all $t \leq d(v)$. Let $t_0 \leq d(v)$ be minimal such that $\tau$ leaves $e(v)$ time units of idle time in $[0, t_0]$ when $\vec{\pi}$ 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 6.2.1, so Condition (6.2) cannot hold for this particular $t_0$, leading to a contradiction. \hfill \blacksquare

Note that in Condition (6.1) it is sufficient to only test all integers $t \leq d(v)$ since request functions only change at integer points. There are two reasons:

---

2This time point is also called the response time of the job. We deal with response times in greater detail in Chapter 7.
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 this chapter, functions only need to be evaluated at integer points (as throughout most parts of this thesis).

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 $\Pi$ before, we write $RF(\tau)$ for all combinations of tasks, i.e., $RF(\tau) = RF(T_1) \times \ldots \times RF(T_N)$ and $\bar{r}f = (rf(T_1),\ldots,rf(T_N))$ for elements of $RF(\tau)$. With this notation, Condition (6.1) is equivalent to

$$\forall \bar{r}f \in RF(\tau) : \exists t \leq d(v) : e(v) + \sum_{T \in \tau} rf(T)(t) \leq t.$$  

(6.3)

### 6.2.2 Critical Request Functions

The test in Condition (6.3) is already finite, since $RF(\tau)$ can be effectively and finitely enumerated, but the number of request functions per task is exponential in $d$. We will see that only a small fraction of them is actually relevant. Consider two request functions $rf$ and $rf'$ such that $rf(t) \geq rf'(t)$ for all $t$ in $[0,d]$. If Condition (6.3) 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 6.2.3.** For two request functions $rf$ and $rf'$ on domain $[0,d]$, we say that $rf$ dominates $rf'$, written $rf \succeq rf'$, if and only if

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

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

**Example 6.2.4.** As an example, we take again the request functions on $[0,30]$ in Figure 6.3 on page 92. Note that $rf_\pi$ has at each point a value at least as large as $rf_{\pi'}$. Therefore we have $rf_\pi \succeq rf_{\pi'}$. The same holds with $rf_{\pi'}$, i.e., $rf_{\pi'} \succeq rf_{\pi'}$. In fact, $rf_\pi$ and $rf_{\pi'}$ are critical request functions for this task.
Let \( RF^*(T) \) denote the (unique) set of critical request functions for \( T \) and let \( RF^*(\tau) \) be defined analogously. Then Condition (6.3) is equivalent to the following which is quantifying only over all combinations \( RF^*(\tau) \) of critical request functions instead of \( RF(\tau) \).

\[
\forall \bar{rf} \in RF^*(\tau) : \exists t \leq d(v) : e(v) + \sum_{T \in \tau} rf^{(T)}(t) \leq t.
\] (6.4)

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 (6.4) needs to be checked, despite the theoretically exponential size of all \( RF^*(T) \) in the worst case.

### 6.2.3 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 6.3 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 the iterative graph exploration technique presented in Chapter 3. Recall that this technique used demand triples as path abstractions in order to efficiently compute the set of demand pairs from which the demand-bound function could be derived. For our problem at hand, we use request functions as path abstractions in the same graph exploration approach. Recall that the idea is to start with all 0-paths in the graph, i.e., paths containing just a single vertex, and to iteratively extend 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.

### 6.2.4 The Naïve Algorithm

We summarize this section by presenting a naïve first version of the complete algorithm in Figures 6.4 to 6.7, based on Lemmas 6.1.3 to 6.1.5 and Theorem 6.2.2. Assumed is a function \( \text{generate-rfs}(T) \) 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 6.2.3. We further assume that vertices have been marked as critical and implicitly update these markings in \( SP\text{-feasible}(\tau) \).

Note that \( SP\text{-schedulable}(\tau, Pr) \) makes \( O(\|\tau\|) \) calls to \( lp\text{-feasible}(T, \tau) \), compared to \( SP\text{-feasible}(\tau) \) making \( O(\|\tau\|^2) \) such calls. Thus, the run-time difference is only about a factor linear in the number of tasks.
function schedulable(v, τ):
1: for all T ∈ τ do
2: RF∗(T) ← generate-rfs(T)
3: end for
4: for all rf ∈ RF∗(τ) do
5: if ∀t ≤ d(v): e(v) + ∑T∈τ rf(T)(t) > t then
6: return false
7: end if
8: end for
9: return true

Figure 6.4. Algorithm for schedulability of a vertex v with interference set τ.

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

Figure 6.5. Algorithm for lowest-priority feasibility of a task T ∈ 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 naïve combinatorial test with our proposed iterative approach using combinatorial abstraction refinement.

6.3 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. But if a combination signals nonschedulability, 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 com-
function $SP$-schedulable$(\tau, Pr)$:
1: for all $T \in \tau$ do
2: if not $lp$-feasible$(T, \tau \leq T)$ then
3: return false
4: end if
5: end for
6: return true

Figure 6.6. Algorithm for SP schedulability of a task set $\tau$ with priorities $Pr$.

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

Figure 6.7. Algorithm for SP feasibility of a task set $\tau$.

Arkoglis and coauthors worked with 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.

### 6.3.1 Abstract Request Functions

We introduce an abstraction of a set of request functions by taking their pointwise maximum.

**Definition 6.3.1.** We call $rf$ a concrete request function if it is derived from a path $\pi$ in a graph $G(T)$ as in Definition 6.2.1.

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 $rf = rf_1 \sqcup \ldots \sqcup rf_k$. \[\square\]
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 $RF^*(T)$ 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

$$\exists t \leq d(v) : e(v) + \sum_{T \in \tau} mrf(T)(t) \leq t. \tag{6.5}$$

This holds because Condition (6.5) implies Condition (6.4) from Section 6.2.2 (page 95). The test is much more efficient since it uses only one combination of (now abstract) request functions instead of exponentially many.

However, Condition (6.5) is over-approximate. If it is satisfied, vertex $v$ is indeed schedulable, but if it fails, $v$ may still be schedulable. See Figure 6.8 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 (6.4) and (6.5) 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.

### 6.3.2 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 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 (6.5) but still more efficient than Condition (6.4). 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 \( RF^*(T) \). 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 6.3.3 below). 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 6.9 illustrates the abstraction tree.

![Abstraction Tree](image)

Figure 6.9. Request function abstraction tree for request functions of task \( T \) in Figure 2.2. 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. 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, exactly as in Condition (6.5). 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 trough the trees down to the tree leaves since schedulable subtrees are avoided.

\[ \text{The point-wise maximum on request functions and the dominance relation from Definition 6.2.3 on page 94 are a join-semilattice} (\succeq, \sqcup) \text{ on the request functions for each task. These semilattices are the core structure of our abstraction refinement technique.} \]
6.3.3 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 a greater diversification of scenarios 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 6.3.2.** For two request functions $rf$ and $rf'$ we define their distance on domain $[0,d]$ as

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

We choose to introduce a weighting factor $\alpha$ 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 $\alpha = \sqrt[10]{0.1}$ with $\alpha^0 = 1$ and $\alpha^d = 0.1$.

6.3.4 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 6.2 is the implementation of $\text{ schedulable}(v, \tau)$ which we replace with $\text{ schedulable-car}(v, \tau)$ in Figure 6.10.
The implementation assumes a function $generate-mrf(T,d)$ which generates the abstraction tree for critical request functions on $[0,d]$ and returns the tree root, i.e., the most abstract request function for $T$. We further assume a function $refine(rf)$ 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 $v$.

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

Figure 6.10. Improved algorithm based on combinatorial abstraction refinement for schedulability of a vertex $v$ with interference set $τ$.

6.4 General Formulation

We conclude this chapter with 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 $\succ_i$ 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 $\succeq$ on the product space:
  \[
  a_1 \succeq_1 a'_1 \land \ldots \land a_N \succeq_N a'_N \iff \langle a_1, \ldots, a_N \rangle \succeq \langle a'_1, \ldots, a'_N \rangle.
  \]
  This implies that all elements from $S_1 \times \ldots \times S_N$ are the minimal elements of $\succeq$ and that $\langle \top_1, \ldots, \top_N \rangle$ is the greatest element.

- There is a predicate $P : A_1 \times \ldots \times A_N \rightarrow \{true, false\}$.
  For concrete combinations, this predicate distinguishes positive from negative instances.

- The predicate is monotonic with respect to the partial order, that is:
  \[
  [P(\langle a_1, \ldots, a_N \rangle) \land \langle a_1, \ldots, a_N \rangle \succeq \langle a'_1, \ldots, a'_N \rangle] \Rightarrow P(\langle a'_1, \ldots, a'_N \rangle).
  \]

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 $\text{coNP}$ 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 6.11. As for its instance used for testing schedulability in Section 6.3.4, we assume a few associated auxiliary functions.

- A data structure $\text{store}$ used for storing and retrieving tuples from the abstract product domain $A_1 \times \ldots \times A_N$. The store needs to support functions $\text{add}$ for storing and $\text{pop}$ for retrieving tuples. A function $\text{isempty}$ indicates whether the store is empty.

- A function $\text{refine}$ takes one tuple $\langle a_1, \ldots, a_N \rangle$ as its argument and returns a set of tuples $\{\langle b_1, \ldots, b_N \rangle\}$ which, intuitively, is a complete set of direct descendants in $\succeq$. Formally:
1. Each \( \langle b_1, \ldots, b_N \rangle_i \) is dominated by \( \langle a_1, \ldots, a_N \rangle \), i.e.,
\[
\forall i : \langle a_1, \ldots, a_N \rangle \succeq \langle b_1, \ldots, b_N \rangle_i.
\]

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.,
\[
\langle a_1, \ldots, a_N \rangle \succeq \langle s_1, \ldots, s_N \rangle,
\]
there is one of the new tuples \( \langle b_1, \ldots, b_N \rangle_i \) dominating it, i.e.,
\[
\exists i : \langle b_1, \ldots, b_N \rangle_i \succeq \langle s_1, \ldots, s_N \rangle.
\]

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 \( \succeq_i \) can be represented as a tree, \textit{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 \textit{isabstract} testing whether a tuple \( \langle a_1, \ldots, a_N \rangle \) is concrete or not, i.e.,
\[
isabstract(\langle a_1, \ldots, a_N \rangle) \iff \langle a_1, \ldots, a_N \rangle \notin S_1 \times \ldots \times S_N.
\]

Clearly, the method presented in Section 6.3 is an instance of this general formulation in which \( S_i = RF^* (T_i) \), predicate \( P \) is the existence of an idle instant and all partial orders in all components are given by the abstraction trees.

\begin{verbatim}
function find-negative(P):
    1: store ← \emptyset
    2: store.add(\langle \top, \ldots, \top \rangle)
    3: while not store.isempty() do
        4: \langle a_1, \ldots, a_N \rangle ← store.pop()
        5: if \neg P(\langle a_1, \ldots, a_N \rangle) then
            6: if isabstract(\langle a_1, \ldots, a_N \rangle) then
                7: store.add(refine(\langle a_1, \ldots, a_N \rangle))
            8: else
                9: return \langle a_1, \ldots, a_N \rangle
        10: end if
    11: end if
    12: end while
    13: return false
\end{verbatim}

\textit{Figure 6.11.} 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 \textit{false}.
7. Response-Time Analysis

From the qualitative analysis in previous chapters, classifying task sets as schedulable or unschedulable, we now switch to a quantitative analysis by computing worst-case response times of jobs. The response time of a job is the difference between its release and its finish time. It can be used for assessing schedulability by comparing its worst case to job deadlines, but has other potential uses. Take for example a distributed real-time system where task releases are triggered by other tasks finishing. Bounds on invocation jitters can be tightened by using knowledge about response times of triggering tasks.

This chapter presents a response-time analysis method for SP and EDF schedulers based on the combinatorial abstraction refinement framework described in Chapter 6. The method roots on an exact characterization of response times for DRT task systems. For SP schedulers, the analysis is very similar to the SP schedulability test presented in Chapter 6, since the schedulability test basically computes an upper bound on the response time. The method needs to be refined since the schedulability test stops already as soon as the response time bound is less than the deadline, but we want to derive an exact value. For EDF schedulers, the response time characterization needs to be extended significantly since the approach of only focusing on a job’s scheduling window is not safe in the EDF setting. Further, the dynamically changing priorities yield an additional challenge when quantifying job interferences.

7.1 Preliminaries

We define the worst-case response time as a property of a vertex, representing all jobs which it releases.

**Definition 7.1.1 (Worst-Case Response Time).** Given a task set $\tau$ and a scheduler $Sch$, the response time $R_{Sch}(v)$ of a vertex $v \in V(T)$ for a task $T \in \tau$ is the maximal time between release and finish of any job corresponding to $v$ for all job sequences generated by $\tau$ when scheduled with $Sch$. □

We briefly review the classic response-time analysis for periodic tasks and SP scheduling [28]. Consider a set $\tau$ of periodic tasks, i.e., for each $T \in \tau$ its graph $G(T)$ contains only a single vertex $v$ and a self-loop $(v,v)$. We write $E(T)$ for $e(v)$, $P(T)$ for $p(v,v)$ and $T' > T$ for a task $T'$ with higher priority.
than $T$. The response time $R(T) = R_{SP}(v)$ can be computed with the following equation.

$$R(T) = \min \left\{ t \geq 0 \mid E(T) + \sum_{T' > T} \left\lceil \frac{t}{P(T')} \right\rceil E(T') \leq t \right\}$$ \hspace{1cm} (7.1)

The term $\left\lceil \frac{t}{P(T')} \right\rceil E(T')$ characterizes the interference of each higher-priority task $T'$ in a time interval of size $t$. It can be interpreted as a (left-continuous) step function in $t$. Intuitively, the minimum searches for the first time point $t$ in which the job under consideration and all interfering work load of higher-priority tasks have finished their execution. A key observation is that the response time of the job released by $v$ is maximized when all other tasks release a job at the same time instant as $v$, with subsequent job releases as early as possible. This situation is usually called the critical instant. We will generalize the analysis approach to DRT task systems.

### 7.2 RTA for SP Scheduling

For a generalization of the above method for periodic tasks to the DRT task model, we need to cope with tasks represented by arbitrary graphs containing different types of jobs, in contrast to the relatively simple structure of periodic tasks. Different combinations of paths from all interfering tasks can clearly lead to different response times of an analyzed vertex.

**Example 7.2.1.** Consider paths $\pi = (v_4, v_1, v_2)$ and $\pi' = (v_5, v_4, v_1)$ in $G(T)$ of the example task from Figure 6.2 on page 92. Figure 7.1 illustrates the response time of a vertex $v$ with $e(v) = 9$ and $d(v) = 30$ when experiencing interference from both paths. In the upper scenario where $\pi$ causes interference, $v$ has a response time of 18. In the lower one with $\pi'$ causing the interference, the response time is 19 which is in fact the worst-case response time.  

### 7.2.1 Characterization using Request Functions

We use the same approach as in Chapter 6 by abstracting paths using request functions, see Definition 6.2.1 on page 92. Similar to Theorem 6.2.2 (page 93) for characterizing schedulability, we can use request functions to characterize the response time of a vertex by observing that it suffices to focus on the scheduling windows of the jobs it releases\(^1\).

\(^1\)We assume schedulability of task sets in order to simplify presentation. Due to this assumption, a job is guaranteed to not experience any interference from other jobs of the same task. Using a busy window extension technique as below for EDF analysis, the method can be generalized to include tardy jobs.

106
\[ R_{SP}(v, \pi) = 18 \]

\[ R_{SP}(v, \pi') = 19 \]

**Figure 7.1.** Simulated synchronous arrival sequences in the analysis of a job \( v \) with \( e(v) = 9 \) and \( d(v) = 30 \). Example 7.2.1 gives details about the interfering paths.

Let \( \bar{\pi} \) denote a tuple \( \left( \pi(T_1), \pi(T_2), \ldots \right) \) of paths through \( G(T_1), G(T_2), \ldots \) and \( \Pi(\tau) = \Pi(T_1) \times \Pi(T_2) \times \ldots \) its domain. That is, \( \Pi(T) \) is the set of all paths in \( G(T) \) for each task \( T \). We can express the response time of a vertex \( v \) for this particular combination of paths as follows.

\[
R_{SP}(v, \bar{\pi}) := \min \left\{ t \geq 0 \mid e(v) + \sum_{T' > T} rf_{\pi(T')}(t) \leq t \right\} \tag{7.2}
\]

It is easy to see that this is a generalization of the case for periodic tasks in Equation (7.1) by noting that a periodic task \( T \) is uniquely represented by \( rf(t) = \left\lceil t/P(T) \right\rceil E(T) \), cf. Equation (7.1).

**Example 7.2.2.** Consider again the response-time scenario of a vertex \( v \) with \( e(v) = 9 \) and \( d(v) = 30 \) from Figure 7.1. The same scenario is expressed using request functions in Figure 7.2 on page 108. Note that the worst-case response time can easily be retrieved graphically by considering intersections with the diagonal.

The overall worst-case response time of \( v \) is the maximum over all such path combinations \( \bar{\pi} \).

\[
R_{SP}(v) = \max_{\bar{\pi} \in \Pi(\tau)} R_{SP}(v, \bar{\pi}) \tag{7.3}
\]
Figure 7.2. Response-time scenario from Example 7.2.1 on page 106. Paths are taken from $G(T)$ in Figure 6.2 with $\pi = (v_4, v_1, v_2)$ and $\pi' = (v_5, v_4, v_1)$, cf. Figure 7.1. The derived response times $R_{SP}(v, \pi) = 18$ and $R_{SP}(v, \pi') = 19$ coincide with those observed in the SAS simulation in Figure 7.1.

This condition gives a formal characterization of $R_{SP}(v)$. We have seen in our SP schedulability method in Chapter 6 that the set of request functions per task that need to be considered can be substantially lowered by only focusing on critical request functions, cf. Definition 6.2.3 on page 94.

We can express the worst-case response time $R_{SP}(v)$ directly in terms of critical request functions. Let $\bar{rf}$ denote a tuple $\left\{ rf(T_1), rf(T_2), \ldots \right\}$ of request functions. For each $T$, let $RF^*(T)$ denote the set of critical request functions on domain $[0, d]$ and $RF^*(\tau) = RF^*(T_1) \times RF^*(T_2) \times \ldots$ all their combinations. Using this notation, Equations (7.2) and (7.3) can be expressed as follows.

$$R_{SP}(v, \bar{rf}) := \min \left\{ t \geq 0 \mid e(v) + \sum_{T' > T} rf(T')(t) \leq t \right\} \quad (7.4)$$

$$R_{SP}(v) = \max_{\bar{rf} \in RF^*(\tau)} R_{SP}(v, \bar{rf}) \quad (7.5)$$

Condition (7.5) leads to a combinatorial explosion which we solve with an extension of the combinatorial abstraction refinement technique from Chapter 6, described in the following section.

### 7.2.2 Applying Combinatorial Abstraction Refinement

Recall that the key idea of the combinatorial abstraction refinement scheme is that several request functions can be represented by a single one, called abstract request function, by taking their point-wise maximum. Using abstract request functions results in an over-approximate response time. By step-wise refinement of the abstraction using abstraction trees, the result can be made more and more precise until it eventually is not over-approximate anymore.
This leads to a dramatically improved performance without compromising precision of the result.

Consider for each $T$ the most abstract request function $mrf^{(T)}$ from Section 6.3.1, defined as the point-wise maximum of all its concrete request functions $RF^*(T)$. It can be used for a response-time over-approximation:

$$R_{SP}(v) \leq \min \left\{ t \geq 0 \mid e(v) + \sum_{T' > T} mrf^{(T')}(t) \leq t \right\}$$

Note that the RHS is not using a maximum operator over a set of combinations. Only one combination of (abstract) request functions is used, the $mrf^{(T')}$ for every task $T'$ of higher priority than $T$. This is very efficient, but does not necessarily compute a precise response time as Figure 7.3 demonstrates.

The idea is to start with such an over-approximate result and refine it step-by-step. This is done just as for the schedulability test in Chapter 6, except that we do not stop the refinement procedure when the current upper bound does not exceed the deadline anymore. Instead, the procedure continues to refine until a leaf combination is found with a response time equal to the current upper bound. We give now the details.

**Refinement Procedure**

The procedure starts with just one tuple $\bar{rf}$ of abstract request functions, i.e., $(mrf^{(T_1)}, mrf^{(T_2)}, \ldots)$ for computing a first over-approximate response time $R_{SP}(v, \bar{rf})$. In each step, one abstract request function $rf$ in the tuple is replaced by refinements $rf'$ and $rf''$ with

$$rf \succ rf', \quad rf \succ rf'', \quad rf = rf' \sqcup rf''.$$ 

In other words, $rf$ is replaced by its two child nodes in the corresponding abstraction tree. As before, we call this a *split* of $rf$, leading to two new tuples.
of request functions $\bar{rf}'$ and $\bar{rf}''$ which are identical to $\bar{rf}$ except that $rf$ is exchanged for $rf'$ and $rf''$, respectively. The new response times $R_{SP}(v, \bar{rf}')$ and $R_{SP}(v, \bar{rf}'')$ are still over-approximate, but more precise than the first one. Of the current set of tuples, we always split the one with the largest response time over-approximation. The splitting results in more and more tuples in the current set and eventually, some of them will consist of only concrete request functions. The procedure ends as soon as one of these tuples of concrete request functions leads to the maximal response time of any tuple in the current set. We give our refinement procedure for computing static priority response times in Figure 7.4.

function compute-$R_{SP}(v)$:
1: store $\leftarrow \emptyset$
2: for all $T' > T(v)$ do
3: \( rf(T') \leftarrow \text{generate-mrf}(T', d(v)) \)
4: end for
5: store.add($\bar{rf}, R_{SP}(v, \bar{rf})$)
6: while isabstract(store.head) do
7: $\bar{rf} \leftarrow$ store.pophead()
8: ($\bar{rf}', \bar{rf}'') \leftarrow \text{refine}(\bar{rf})$
9: store.add($\bar{rf}', R_{SP}(v, \bar{rf}')$)
10: store.add($\bar{rf}'', R_{SP}(v, \bar{rf}'')$)
11: end while
12: return $R_{SP}(v, \text{store.head})$

Figure 7.4. Algorithm for computing worst-case response time $R_{SP}(v)$ of a vertex $v$.

The procedure in Figure 7.4 uses a priority queue $\text{store}$ for storing tuples of request functions. This data structure has to support two functions to manipulate its contents. A function $\text{add}(\text{value}, \text{priority})$ adds a new value to the store which is associated with a priority. As a result, the head of $\text{store}$, accessed via $\text{store.head}$, is expected to always contain the value with the highest priority and can be retrieved with the $\text{pophead}()$ function. We further assume a function $\text{generate-mrf}(T, d)$ which computes the abstraction tree for task $T$ and returns $\text{mrf}(T)$ on domain $[0, d]$. Finally, as in the algorithm in Section 6.3.4, we assume a function $\text{refine}(\bar{rf})$ 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).

Correctness
The worst-case response time returned by procedure $\text{compute-R}_{SP}(v)$ is exact. The intuition is that the head of the priority queue is a tuple $\bar{rf}$ for which $R_{SP}(v, \bar{rf})$ is always a safe upper bound for the response time of $v$. As long as this tuple still contains an abstract request function for a task, this upper bound
may be strictly over-approximate and thus needs to be refined. However, if it only contains concrete request functions, a set of paths trough all graphs is found which leads to this particular response time. We summarize this insight in the following theorem.

**Theorem 7.2.3.** *The return value of function compute-\(R_{SP}(v)\) in Figure 7.4 is equal to the worst-case response time \(R_{SP}(v)\) of \(v\).* □

**Proof.** We prove the correctness by showing a loop invariant and establish the result as the conjunction of loop invariant and negated loop condition.

**Loop Invariant.** The algorithm in Figure 7.4 has the following loop invariant which holds at the start of its main loop in line 6. Let \(\tau_{\text{high}}\) denote the set of all \(T'\) with \(T' > T(v)\).

\[
\forall \bar{\pi} \in \Pi(\tau_{\text{high}}) : \exists \bar{r}f \in \text{store} : R_{SP}(v, \bar{\pi}) \leq R_{SP}(v, \bar{r}f).
\]

In other words, the associated request functions of each combination of paths through the graphs of \(\tau_{\text{high}}\) are always dominated by some currently stored tuple of (possibly abstract) request functions. Initially, \(\text{store}\) only contains one tuple of all most abstract request functions which by construction of the abstraction trees must dominate all combinations of critical request functions. For the induction step, let \(\bar{\pi} \in \Pi(\tau_{\text{high}})\) and \(\bar{r}f \in \text{store}\) such that \(R_{SP}(v, \bar{\pi}) \leq R_{SP}(v, \bar{r}f)\) at the start of the loop. If \(\bar{r}f\) is not the tuple removed by the \(\text{pophead}()\) call, then the invariant trivially holds at the end of the loop body. Otherwise, \(\bar{r}f\) is split into \(\bar{r}f'\) and \(\bar{r}f''\). The only difference between these two and \(\bar{r}f\) is that one of the request functions in \(\bar{r}f\) has been replaced by its child nodes in its abstraction tree. Since \(\bar{\pi}\) must be represented by some combination of child nodes all of which are in the subtrees of \(\bar{r}f\), either \(\bar{r}f'\) or \(\bar{r}f''\) still dominate this combination, so either \(R_{SP}(v, \bar{\pi}) \leq R_{SP}(v, \bar{r}f')\) or \(R_{SP}(v, \bar{\pi}) \leq R_{SP}(v, \bar{r}f'')\). Both \(\bar{r}f'\) and \(\bar{r}f''\) are added to the store, so the invariant is restored.

**Termination.** When the loop terminates, we show that

\[
\exists \bar{\pi} \in \Pi(\tau_{\text{high}}) : R_{SP}(v, \bar{\pi}) = R_{SP}(v, \text{store}.\text{head}).
\]

Indeed, since the loop condition does not hold anymore, i.e., \(\text{store}.\text{head}\) only contains concrete request functions, there must be a combination of paths through the graphs which leads to this response time.

Finally, it follows from the loop invariant that no combination of paths can cause a strictly larger response time than the return value, since \(\text{store}.\text{head}\) maximizes the response time over all tuples in \(\text{store}\). Therefore, the return value is indeed equal to \(R_{SP}(v)\).
7.3 RTA for EDF Scheduling

In this section we adjust the above response-time analysis method to the setting of an EDF scheduler. The major difference is the response-time computation given a combination of paths (cf. Equation (7.4) on page 108) which needs to be changed and extended. We use a characterization inspired by [26]. However, we will see that the basic refinement framework from above can be reused in order to obtain exact results from initial over-approximate estimations.

7.3.1 Workload Functions

In SP scheduling, the static priority order of tasks directly determines which jobs may cause interference to each other: each job of a task of higher priority may preempt any job from a task of lower priority and thus delay its execution and increase its response time. There is never any interference in the opposite direction, from low to high priorities. Neither do jobs of the same task cause interference to each other in a setting of static task priorities and constrained deadlines. Based on this property, we were able to use request functions in order to quantify interference from tasks of higher priority.

In EDF scheduling, the situation changes. When analyzing the response time of a particular job, certain jobs from other tasks may cause interference but others do not, at least not in the same release scenario.

**Example 7.3.1.** Consider the schedule in Figure 7.5(a). Illustrated is the response-time analysis of v with \( e(v) = 20 \) and \( d(v) = 37 \) with an interfering periodic task. In this scenario, only the first two jobs of the interfering task \( T_1 \) do interfere with v under EDF since the third one has a later deadline than v and therefore lower priority. □

In order to quantify this interference, we would like to use a function similar to a request function, but it should be counting only jobs with higher priority, i.e., earlier absolute deadline. Thus, such a function needs to consider deadlines instead of release times. We define a demand function for a path \( \pi \) to return the accumulated execution requirement of all jobs of \( \pi \) that are released and have a deadline during the first \( t \) time units.

**Definition 7.3.2 (Demand Function).** For a path \( \pi = (\pi_0, \ldots, \pi_l) \) through the graph \( G(T) \) of a task T, we define its demand function as

\[
 df_{\pi}(t) := \max \{ e(\pi') \mid \pi' \text{ is prefix of } \pi \text{ and } d(\pi') \leq t \}
\]

where \( d(\pi) := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + d(\pi_l) \) and \( e(\pi) \) as before. □

However, using just a demand function does still not exactly quantify the interference since it may also be over-approximate.
Figure 7.5. Two different scenarios for a job $v$ with $\langle e(v), d(v) \rangle = \langle 20, 37 \rangle$ to experience interference from periodic tasks $T_1$ or $T_2$. Not all of their jobs that are released during the scheduling window of $v$ do actually cause interference.

**Example 7.3.3.** Consider the schedule in Figure 7.5(b). In this second scenario, only the first three jobs of the interfering task $T_2$ do interfere with $v$. Unlike the fifth job, the fourth one has its deadline before $d(v)$ and is therefore included in a demand function. However, it also does not cause interference since it is actually released after the finish time of $v$. Thus, only the first three jobs should be counted when trying to exactly quantify the maximal interference.

The solution is to define for each path a function which counts only jobs with a deadline earlier than the deadline of $v$ but also with a release time earlier than the finish time of $v$. A workload function has therefore a two-dimensional domain. Intuitively, the first argument $t$ represents the supposed finish time of $v$ and jobs from $\pi$ that are counted should be released earlier than $t$. The second argument $t'$ represents the deadline of $v$ and jobs from $\pi$ that are counted should have their deadline latest at $t'$.

**Definition 7.3.4 (Workload Function).** For a path $\pi = (\pi_0, \ldots, \pi_l)$ through the graph $G(T)$ of a task $T$, we define its workload function as

$$wf_\pi(t, t') := \max \{e(\pi') \mid \pi' \text{ is prefix of } \pi, p(\pi') < t \text{ and } d(\pi') \leq t'\}.\qed$$

**Example 7.3.5.** Consider the periodic task $T_1$ in Figure 7.5(a). We have $wf(30, 37) = 10$ since two jobs of 5 execution time units each are released
within 30 time units and have their deadlines within 37 time units. As a consequence, the exact interference that v experiences is 10 time units and it finishes after 30 time units. Note that we do not write a path π as subscript to wf in this case since a periodic task has only one (infinite) path.

As a second example, consider the periodic task T2 in Figure 7.5(b). In this case we have wf(26,37) = 6, counting the first three jobs of T2, since these are the only ones being released within 26 time units and having a deadline within 37 time units. Therefore, v experiences 6 time units of interference and finishes after 26 time units.

Workload functions are a generalization of request functions as well as demand functions, since

\[ rf(\pi)(t) = wf(\pi,\infty) \quad \text{and} \quad df(\pi)(t) = wf(\infty,\pi). \]

Moreover, a workload function can be derived from request and demand functions of the same path, which allows compact representation in memory.

**Lemma 7.3.6.** For a path \( \pi \) through a graph \( G(T) \), the functions \( rf(\pi) \), \( df(\pi) \) and \( wf(\pi) \) are related via

\[ \forall t,t' \geq 0 : wf(\pi,\pi')(t,t') = \min(rf(\pi)(t), df(\pi')(t')). \]

**Proof.** Given \( t,t' \) and a path \( \pi \), let \( \pi'_{rf} \) and \( \pi'_{df} \) be the paths used in the maximum for the computations of \( rf(\pi)(t) \) and \( df(\pi')(t') \), respectively. Both paths are prefixes of \( \pi \) and we have two cases depending on their lengths.

Case 1: \( \pi'_{rf} \) is a prefix of \( \pi'_{df} \). Thus, \( rf(\pi)(t) \leq df(\pi')(t') \). Further, \( d(\pi'_{rf}) \leq d(\pi'_{df}) \) since deadlines are constrained and therefore ordered monotonically. Hence \( wf(\pi,\pi')(t,t') \geq e(\pi'_{rf}) = rf(\pi)(t) \) since \( \pi'_{rf} \) satisfies both conditions in the definition of \( wf \). However, any prefix \( \pi' \) of \( \pi \) longer than \( \pi'_{df} \) has \( p(\pi') \geq t \), therefore \( rf(\pi)(t) \) is the maximal value for \( wf(\pi,\pi')(t,t') \).

Case 2: \( \pi'_{df} \) is a (strict) prefix of \( \pi'_{rf} \), same proof but with swapped roles of \( df \) and \( rf \).

We can now do a first attempt to use workload functions for response time analysis. Consider a vertex \( v \in V(T) \) from a task \( T \) for which we want to compute the worst-case response time with an EDF scheduler and pick a workload function \( wf(T) \) for each task \( T \). We search for the first time instant \( t_f \) at which a job \( v \) does finish its execution. While doing so, we need to consider interference of jobs from other tasks which

1. are released strictly before \( t_f \) and
2. have their deadline latest at \( d(v) \).
The following expression captures this intuition and is very similar to Equation (7.4) on page 108 for SP scheduling.

\[ t_f = \min \left\{ t \geq 0 \mid e(v) + \sum_{T' \neq T} \text{wf}(T')(t, d(v)) \leq t \right\} \]  

(7.6)

### 7.3.2 Busy Window Extension

As we will see now, this \( t_f \) does not correctly capture the worst-case response time of \( v \). The underlying assumption is that a synchronous arrival sequence creates the maximal interference for a job. Unfortunately, this assumption does not hold in the EDF case as the following example shows.

**Example 7.3.7.** 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 vertex \( v_4 \) in task \( T \) from Figure 6.2 on page 92, preempted by two periodic tasks \( T_1 \) and \( T_2 \). We illustrate the situation in Figure 7.6. \( T \) first releases a job from \( v_5 \) at time \( t = 2 \) followed by the job from \( v_4 \) to be analyzed at \( t = 10 \). This situation leads to the worst-case response time \( R_{EDF}(v_4) = 9 \). In an SAS, the job corresponding to \( v_4 \) would finish after already 8 time units. □

![Figure 7.6](image-url)

**Figure 7.6.** 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 \( x = 10 \). Details are discussed in Example 7.3.7.

The issue is that because of the dynamic priorities used by EDF, jobs from the same task can cause (indirect) interference to each other. In order to solve this problem, we use the common busy window extension technique. The idea
is as follows. The worst-case response time of \( v \) 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 worst-case response time of \( v \), but an upper bound can be given by computing the maximal size of any busy window\(^2\) for \( \tau \). With this upper bound, all possibilities can be enumerated.

We now give details of this procedure. 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 7.6. For each integer \( x \geq 0 \), 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, i.e., corresponding to a path through \( G(T) \) ending in \( v \). This construction maximizes the interference experienced by \( v \) for a particular \( x \). In Figure 7.6 this is the case for \( x = 10 \) where \( T \) is following path \((v_5, v_4)\). In order to find the worst-case response time, all \( x \) are enumerated up to the size \( L \) of the maximal busy window.

Formally, we can use workload functions as before to describe the interference to \( v \) by tasks \( T' \neq T \). The value of \( \text{wf}_\pi(x + t, x + d(v)) \) for a path \( \pi \) captures the interference of all jobs along path \( \pi \) that

1. are released within the extended busy window before a supposed finish time \( x + t \) of \( v \), but
2. still have their deadline latest at \( x + d(v) \) so that they actually interfere with \( v \) because of having a higher priority under EDF.

In addition to that, we also need to have a way of describing the 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 demand function which is defined like a demand function but ensures that the last vertex of a path is always included in the workload.

**Definition 7.3.8** (Suffix Demand Function). For a path \( \pi = (\pi_0, \ldots, \pi_l) \) in the graph \( G(T) \) of a task \( T \), we define its suffix demand function as

\[
df_{\pi}^{\text{fx}}(t) := \max \{ e(\pi') \mid \pi' \text{ is suffix of } \pi \text{ and } d(\pi') \leq t \}.
\]

**Example 7.3.9.** We give an example in Figure 7.7.

\(^{2}\) A bound for the size of the maximal busy window can be easily computed by finding smallest interval size \( t \) for which \( \sum_{T \in \tau} \text{mrf}(T)(t) \leq t \).
Figure 7.7. Example of a suffix demand function on \([0, 30]\) for \(\pi = (v_5, v_5, v_4)\) from \(G(T)\) in Figure 2.2. We also show the job sequence in a graphical schedule, with time 0 being the deadline for the job from \(v_4\). This illustrates that \(df_{\text{sfx}}^\pi\) builds backwards from the deadline.

We are now finally ready to give the full method for computing the worst-case EDF response time of a vertex \(v \in V(T)\) from a task \(T\). Let \(\bar{wf}\) be a vector containing

- a workload function \(wf(T')\) for each task \(T' \neq T\) describing its workload interfering with \(T\), and
- for task \(T\) a function \(f(t, t')\) defined as
  \[
  f(t, t') := \max(df_{\text{sfx}}^\pi(t'), e(v)).
  \]

This function expresses the workload of \(T\) and always contains the job of \(v\) itself. All jobs released by \(T\) are always implicitly released before time \(t\) which therefore is discarded, i.e., just a dummy argument. The role of this function is just to unify notation so that we can represent all functions in vector \(\bar{wf}\) containing only functions in two arguments.

Using \(\bar{wf}\) we can express the response time \(R_{EDF}(v, \bar{wf}, x)\) of \(v\) for this particular \(\bar{wf}\) and a particular \(x\). The worst-case response time \(R_{EDF}(v)\) of \(v\) is derived by maximizing over all \(x\) and \(\bar{wf}\).

\[
R_{EDF}(v, \bar{wf}, x) := \min \left\{ t \geq 0 \mid \sum_{T \in \tau} wf(T)(x+t, x+d(v)) \leq x+t \right\} \tag{7.7}
\]

\[
R_{EDF}(v) = \max_{\bar{wf}} \max_{x \leq L} R_{EDF}(v, \bar{wf}, x) \tag{7.8}
\]

7.3.3 Refinement Procedure

As for the SP scheduler case, enumerating all combinations of workload functions and suffix demand functions leads to combinatorial explosion. However, we can employ the same combinatorial abstraction refinement technique as
in Section 7.2. Abstract workload functions and suffix demand functions can be defined just as in the case of request functions. Their abstraction trees are also built in a similar way. They can be stored efficiently since for abstract workload functions, we may choose to just store pairs of abstract request and demand functions instead. Because of Lemma 7.3.6, these can be used to derive corresponding abstract workload functions using the minimum operator.

In summary, the full algorithm is given in Figure 7.8.

function compute-$R_{EDF}$($v$) :
1: store $\leftarrow \emptyset$
2: $wf(T(v)) \leftarrow \text{generate-}mdfsfx(T(v), L + d(v))$
3: for all $T' \neq T(v)$ do
4: $wf(T') \leftarrow \text{generate-}mwf(T', L + d(v))$
5: end for
6: store.add($wf, \max_{x \leq L} R_{EDF}(v, wf, x))$
7: while isabstract(store.head) do
8: $wf \leftarrow \text{store.pophead}()$
9: ($wf', wf''$) $\leftarrow \text{refine}(wf)$
10: store.add($wf', \max_{x \leq L} R_{EDF}(v, wf', x))$
11: store.add($wf'', \max_{x \leq L} R_{EDF}(v, wf'', x))$
12: end while
13: return $\max_{x \leq L} R_{EDF}(v, \text{store.head}, x)$

Figure 7.8. Algorithm for computing worst-case response time $R_{EDF}(v)$ of a vertex $v$.

The structure is very similar to the algorithm for SP scheduling in Section 7.2. It is assumed that $L$ is the size of the maximal busy window. Further, functions $\text{generate-}mwf(T, d)$ and $\text{generate-}mdfsfx(T, d)$ compute abstraction trees for a task $T$ on domain $[0, d]$ for all workload functions and suffix demand functions, respectively. Efficient implementations are similar to those for request functions, cf. Section 6.2.3 on page 95. They return the tree roots to be used for the first over-approximate estimate and later refinements, i.e., splitting. The major difference to the SP algorithm is that each response-time computation maximizes over all possible busy window extensions $x \leq L$.

The following theorem states the correctness of our algorithm.

**Theorem 7.3.10.** The return value of function compute-$R_{EDF}$($v$) in Figure 7.8 is equal to the worst-case response time $R_{EDF}(v)$ of $v$. □

**Proof.** Similar to Theorem 7.2.3 on page 111 for SP scheduling via induction over the corresponding loop invariant. The essential difference is the response-time computation for tuples $wf$ which follows from the discussions earlier in this section. ■
7.3.4 Optimizations

Our algorithm for computing the worst-case response time under EDF in this section is slower than the one we present in Section 7.2 for SP. One of the reasons is that for any given combination \( \bar{wf} \) of (abstract) workload functions, all \( x \leq L \) need to be enumerated and \( R_{EDF}(v, \bar{wf}, x) \) needs to be computed via Equation (7.7) again for each \( x \). Evaluating Equation (7.7) so many times can be costly if \( L \) is large. Thus, we employ an optimization that allows to skip a large fraction of possible values for \( x \).

The key idea is to use a simple over-approximation \( R^*_{EDF}(v, \bar{wf}, x) \) of the response time \( R_{EDF}(v, \bar{wf}, x) \).

\[
R^*_{EDF}(v, \bar{wf}, x) := \sum_{T \in \tau} \bar{wf}(T)(t, x + d(v)) - x
\]

Clearly, \( R_{EDF}(v, \bar{wf}, x) \leq R^*_{EDF}(v, \bar{wf}, x) \), since \( R^* \) even counts all jobs from other tasks until the deadline of \( v \), including those released after its finish time, cf. Figure 7.5(b). While searching for the maximal value of \( R_{EDF}(v, \bar{wf}, x) \) over all \( x \), it is only necessary to do an exact evaluation of \( R_{EDF}(v, \bar{wf}, x) \) if \( R^*_{EDF}(v, \bar{wf}, x) \) is strictly larger than the current maximum over those \( x \) already considered. If \( R^*_{EDF}(v, \bar{wf}, x) \) is smaller than the current maximum, then an exact evaluation will not find a new maximum either, so we can directly skip to the next value of \( x \).

Another possible optimization is based on the property of \( rf \) and \( df \) to be step functions. Consider an expression with structure \( f(x) - x \) like the above for \( R^*_{EDF} \). If \( f \) is piecewise constant and we are looking for maximal values of that expression for increasing \( x \), then it is sufficient to evaluate the expression for \( x \) where \( f \) changes. This can be done for the above optimization using \( R^*_{EDF} \), but similar optimizations can be applied to the search for maxima with \( R_{SP} \) and \( R_{EDF} \).
8. Evaluation

We evaluate all analysis methods using randomly generated task sets. We use an implementation in the Python programming language which is suitable for fast prototyping and qualitative comparisons of scaling and other interesting properties. In the evaluation, we focus on four aspects.

**Efficiency.** We investigate analysis methods in terms of scaling properties. That is, if the task set and its parameters grow, how does the run-time of the analysis change? We evaluate EDF and SP feasibility as well as their response-time analyses.

**Run-time Breakdown.** The presented algorithms are usually split into different phases. First, a preparatory stage in which auxiliary structures and values like the utilization or abstraction trees are generated. Second, the actual analysis stage. The evaluation tests how much time of the total analysis run-time is spent in each of the stages and how their ratio changes as task sets grow.

**Refinement Effectiveness.** For the abstraction refinement scheme, we evaluate how effective it is in reducing the necessary work compared to a naïve brute force test of all combinations. For its application to response-time analysis, we further evaluate how much the response-time estimate improves from the initial over-approximate value until the final precise result.

**Acceptance Ratio.** Scheduling algorithms are often compared in terms of acceptance ratio curves, i.e., for each utilization, which ratio of task sets with this particular utilization can successfully be scheduled? We evaluate both EDF and SP scheduling of DRT task sets in terms of acceptance ratio.

### 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 uniformly chosen ratio to the minimal outgoing edge label. Finally, execution time labels are chosen randomly with a uniformly chosen 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. 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.

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 evaluate the feasibility tests and the response-time analysis methods separately.

8.2.1 Feasibility Tests

We compare the pseudo-polynomial algorithm for EDF schedulability presented in Chapter 3 with the combinatorial abstraction refinement method for SP feasibility from Chapter 5. 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 8.1, 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 because of the phase transition between schedulable and unschedulable task sets. Schedulable task sets with high utilization take a long time to analyze since the demand-bound function needs to be evaluated for comparatively many time interval sizes. Unschedulable task sets usually exhibit quite small overloaded time interval sizes. Since we explore task graphs iteratively and are testing the demand-bound function along the way (cf. computation order optimization in Section 3.2.5 on page 40), the test can terminate much earlier in these cases.

The SP test is less sensitive to these effects. It also undergoes a phase transition 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.
8.2.2 Response-Time Analyses

For the response-time analysis methods presented in Chapter 7, we compare the performance of the SP and EDF versions. As for feasibility above, we evaluate how increasing sizes of task sets and their utilization influences the run-time of our method. Since the problem is fundamentally exponential in complexity, we expect also the run-time to increase exponentially. However, the abstraction refinement technique is expected to be effective in hiding the exponential factor which is due to combinatorial explosion. We plot the run-time results in Figure 8.2.

In the SP case in Figure 8.2(a), we note that the run-time is very low even for the largest task sets that are feasible. With our set of parameters, the SP feasibility phase transition ends at around 40%, i.e., task sets with higher utilization are not SP feasible and therefore not eligible for our RTA computation. We see that up to this point, the abstraction refinement technique is effective in hiding most of the exponential increase in run-time.

In the EDF case in Figure 8.2(b), the run-time of our algorithm is longer by about two orders of magnitude. 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. Further, the domain of workload functions increases because of this, leading to increased computational effort for deriving them. Finally, the set of critical workload functions is generally larger than the set of critical request functions (used in the SP case), contributing further to increased complexity of the analysis.
8.3 Analysis Stages

We next evaluate how different parts of the analysis influence the run-time of each method.

8.3.1 EDF Schedulability

The EDF schedulability test introduced in Chapter 3 runs in two stages. First, the utilization is determined in order to compute the counterexample bound. Second, the demand-bound function is computed by successively deriving all demand pairs.

We show the total analysis run-time split into both phases in Figure 8.3. Clearly, the time for computing the utilization grows linearly with the task set size. That is not surprising since this phase is purely local to each task and independent of schedulability behavior. The time for computing the demand-bound function grows rapidly when approaching the phase transition at around 70%. This is because of an increasing bound for a potential counterexample. After the phase transition, counterexamples are found earlier which decreases the time for this phase as discussed above.
8.3.2 SP Feasibility

For static priority feasibility analysis, there are two analysis stages as well. 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 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 8.4 (page 126) 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 exceed two thirds of the analysis time.

8.4 Refinement Effectiveness

The combinatorial abstraction refinement method introduced in Chapter 6 and applied to response-time analysis in Chapter 7 is fundamentally a technique to prune an exponential search space caused by combinatorial explosion. We evaluate the effectiveness of the method in two aspects: how it reduces work and how it improves initial over-approximate response-time results.
8.4.1 Work Reduction

We evaluate how effective the pruning effect is for SP feasibility analysis. (The situation for response-time analysis is very similar.) We captured $10^5$ calls to the iterative abstraction refinement procedure (Figure 6.10 on page 101) and recorded for each call (i) how many tests were executed (Line 8 in Figure 6.10), (ii) how many combinations of concrete request functions there were in total and (iii) its return value. We plot the result in Figure 8.5 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.

8.4.2 Precision Improvement

During the refinement procedure for response-time analysis (Figure 7.8 on page 118), the worst-case response time estimate is improved further and further until the algorithm terminates with a precise value. We evaluate how significant this improvement is. We compare two types of task sets which are created with different sets of parameters. Task sets of type A are created with the same parameters as for the experiments above. Task sets of type B are created with a significantly larger interval of possible edge labels. This impacts also choices of deadlines and WCET labels.

<table>
<thead>
<tr>
<th>Type</th>
<th>Vertices</th>
<th>Fan-out</th>
<th>$p$</th>
<th>$d/p$</th>
<th>$e/d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>[5, 10]</td>
<td>[1, 3]</td>
<td>[100, 300]</td>
<td>[0.5, 1]</td>
<td>[0, 0.07]</td>
</tr>
<tr>
<td>B</td>
<td>[5, 10]</td>
<td>[1, 3]</td>
<td>[10, 300]</td>
<td>[0.5, 1]</td>
<td>[0, 0.07]</td>
</tr>
</tbody>
</table>

126
The idea is to demonstrate that large intra-task differences in labels have an impact on the precision of the initial estimate. For each generated task set of either type, we record the fraction of jobs for which the refinement procedure improved upon the initial over-approximate estimate. Further, for each job where this was the case, we record how large the improvement was. For both types A and B, we run 250 tests each with EDF RTA of task sets with 10 tasks and plot the results in Figure 8.6 (page 128) where each tested task set is represented by a dot or a cross.

A clear difference can be seen between both types of task sets. The precision improvement for type A task sets is both seldom and small. For only up to about 20% of all jobs, the initial estimate is actually over-approximate, and if it is, the precise one is only at most about 5% lower. On the other hand, type B task sets improve the estimate for roughly between 40% and 90% of all jobs for each task, and the average improvement per job is up to about 15% and beyond.

This comparison tells us that our refinement scheme has the potential to significantly improve response-time estimates that were only rough over-approximations with previous methods. This is especially the case if jobs of different types have very different WCET bounds and inter-release delays. Further, since we provide the first tractable exact method, it is now possible to characterize workload for which over-approximate estimates are already rather precise. The more equal WCET bounds and inter-release delays are within tasks, the higher the precision.

Figure 8.5. Tested versus total number of combinations of request functions. Crosses represent schedulable, dots unschedulable cases. Both scales are logarithmic.
8.5 Acceptance Ratio

As a last comparison, we look at acceptance ratios for both EDF and static priority feasibility, shown in Figure 8.7. Note that this comparison is not evaluating the quality of our analysis method. It rather compares the relative scheduling abilities of EDF versus static priority scheduling of DRT tasks. Of course, the phase transitions are sensitive to the choice of parameters for task set creation.

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 any scheduler.
9. Related Workload Models: A Survey

The workload model studied in this thesis finds itself among a variety of models with different expressiveness. This chapter provides a survey on task models to characterize real-time workloads at different levels of abstraction. It covers the classic periodic and sporadic models by Liu and Layland et al., their extensions to describe recurring and branching structures as well as general graph- and automata-based models. The focus is on the precise semantics of the various models and on the solutions and complexity results of the respective feasibility and schedulability analysis problems for preemptable uniprocessors.

9.1 Model Hierarchy

Most workload models described in this survey are part of a model hierarchy in the sense that some are generalizations of others. Intuitively, a model $M_A$ generalizes a model $M_B$, if any task system expressed in terms of $M_B$ has a corresponding task system expressed with $M_A$ such that these two represent the same behavior. The motivation behind such a generalization concept is that more expressive models should preserve the capability of being able to analyze a given system description. That is, if a model is being extended to increase expressiveness, the original model should in some way remain a subclass or special case. However, the concept as such is rather vague, which may lead to disagreements about whether one model is in fact a generalization of another.

Therefore, we choose to formalize the notion of what it means for a model to generalize another one within the semantic framework used in the previous chapters. Recall that we use job sequences to define the semantics of workload models. We first define an equivalence on job sequences.

**Definition 9.1.1** (Job Sequence Equivalence). Two job sequences $\sigma$ and $\sigma'$ are equivalent, written $\sigma \cong \sigma'$, if they are equal after removing all jobs with execution time 0.

This equivalence is useful in order to define equivalent sets of job sequences. Two sets are equivalent if they only contain equivalent job sequences. They express the same workload demands, apart from spurious 0-jobs.

**Definition 9.1.2.** Two sets $S$ and $S'$ of job sequences are equivalent, written $S \cong S'$, if for each $\sigma \in S$, there is $\sigma' \in S'$ with $\sigma \cong \sigma'$, and vice versa.
Since we express workload model semantics as sets of job sequences, we can use this definition in order to give a formal definition of generalization.

**Definition 9.1.3 (Generalization).** A workload model $\mathcal{M}_A$ generalizes a workload model $\mathcal{M}_B$, written $\mathcal{M}_A \succeq \mathcal{M}_B$, if for every task system $\tau_B \in \mathcal{M}_B$ there is a task system $\tau_A \in \mathcal{M}_A$ such that $\llb \tau_A \rrb \simeq \llb \tau_B \rrb$ and $\tau_A$ can be effectively constructed from $\tau_B$ in polynomial time.

This definition allows complexity results for analysis problems to carry over since generalized models can be interpreted as special cases of more general ones. We summarize this in the following proposition.

**Proposition 9.1.4.** If feasibility or schedulability can be decided in pseudo-polynomial time for a workload model $\mathcal{M}$, then they can be decided in pseudo-polynomial time for any $\mathcal{M}'$ with $\mathcal{M} \preceq \mathcal{M}'$.

Further, if feasibility or schedulability are strongly NP- or coNP-hard problems for a workload model $\mathcal{M}$, then they are strongly NP- or coNP-hard for any $\mathcal{M}'$ with $\mathcal{M} \preceq \mathcal{M}'$, respectively.

Using this partial order on workload models, we outline a hierarchy of expressive power in Figure 9.1. An edge $\mathcal{M}_A \longrightarrow \mathcal{M}_B$ represents the generalization relation with the arrow pointing to the more expressive model, i.e., $\mathcal{M}_A \succ \mathcal{M}_B$. The higher a workload model is placed in the hierarchy, the higher
the expressiveness, but also the more expensive feasibility and schedulability analyses.

9.2 From Liu and Layland Tasks to GMF

In this section, we show the development of task systems from the periodic task model to different variants of the Multiframe model, including techniques for their analysis.

9.2.1 Periodic and Sporadic Tasks

The first task model with periodic tasks was introduced 1973 by Liu and Layland [30]. Each periodic task $T = (P, E)$ in a task set $\tau$ is characterized by a pair of two integers: period $P$ and WCET $E$. We can interpret a periodic task as the simplest special case of a DRT task with just one vertex $v$ labeled $\langle e(v), d(v) \rangle = \langle E, P \rangle$ and one edge $(v,v)$ with label $p(v,v) = P$. The job sequences (recall Definition 2.2.1 on page 18) which $T$ generates are of the form $\rho = (J_0, J_1, \ldots)$ containing jobs $J_i = (R_i, e_i, v)$ with release time $R_i$ and execution time $e_i$ such that $R_{i+1} = R_i + P$ and $e_i \leq E$. This means that jobs are released periodically. Further, they have implicit deadlines at the release times of the next job.

A relaxation of this model is to allow jobs to be released at later time points, as long as at least $P$ time units pass between adjacent job releases of the same task. This is called the sporadic task model, introduced by Mok in [33]. Another generalization is to add an explicit deadline $D$ as a third integer to the task definition $T = (P, E, D)$. This is again a special case of DRT tasks and we use the same terminology for indicating a relation between deadline $D$ and period $P$. That is, if $D \leq P$ for all tasks $T \in \tau$ then we say that $\tau$ has constrained deadlines, otherwise it has arbitrary deadlines.

This model has been the basis for many results throughout the years. Liu and Layland give in [30] a simple feasibility test for implicit deadline tasks: defining the utilization $U(\tau)$ of a task set $\tau$ as $U(\tau) := \sum_{T \in \tau} E_i / P_i$, a task set is uniprocessor feasible if and only if $U(\tau) \leq 1$. As in later work, proofs of feasibility are often connected to the Earliest Deadline First (EDF) scheduling algorithm, which uses dynamic priorities and has been shown to be optimal for a large class of workload models on uniprocessor platforms. Recall that because of its optimality, EDF schedulability is equivalent to feasibility (see Theorem 3.1.4 on page 25).

Demand-Bound Functions

For the case of explicit deadlines, Baruah et al. [6] introduced a concept that was later called the demand-bound function. We introduced demand-bound
functions in Section 3.1.1 and just review the main characteristics. For each interval size \( t \) and task \( T \), \( dbf_T(t) \) is the maximum accumulated worst-case execution time of jobs generated by \( T \) in any interval of size \( t \). More specifically, it counts all jobs that have their full scheduling window inside the interval, i.e., release time and deadline. The demand-bound function \( dbf_\tau(t) \) of the whole system \( \tau \) has the property that a task system is feasible if and only if

\[
\forall t \geq 0 : dbf_\tau(t) \leq t. \tag{9.1}
\]

Recall that we used this condition for a test for DRT and EDRT task systems in earlier chapters.

Focussing on sporadic tasks, Baruah et al. show in [6] that \( dbf_\tau(t) \) can be computed with

\[
\text{dbf}_\tau(t) = \sum_{T_i \in \tau} E_i \cdot \max \left\{ 0, \left\lfloor \frac{t-D_i}{P_i} \right\rfloor + 1 \right\}. \tag{9.2}
\]

This closed-form expression is motivated by the observation that periodic tasks lead to a simple form of very regular step-functions. Using this they prove that the feasibility problem is in \( \text{coNP} \). Recently it has been shown by Eisenbrand and Rothvoß [20] that the problem is indeed (weakly) \( \text{coNP} \)-hard for systems with constrained deadlines.

Another contribution of Baruah et al. in [6] was to show that for the case of \( U(\tau) < c \) for some constant \( c \), there is a pseudo-polynomial solution of the schedulability problem, by testing Condition (9.1) for a pseudo-polynomial number of values. The existence of such a constant bound (however close to 1) is a common assumption when approaching this problem since excluding utilizations very close to 1 only rules out very few actual systems.

**Static Priorities**

For static priority schedulers, Liu and Layland show already in [30] that the rate-monotonic priority assignment for implicit deadline tasks is optimal, i.e., tasks with shorter periods have higher priorities. They further give an elegant sufficient schedulability condition by proving that a task set \( \tau \) with \( n \) tasks is schedulable with a static priority scheduler under rate-monotonic priority ordering if

\[
U(\tau) \leq n \cdot \left(2^{1/n} - 1\right). \tag{9.3}
\]

For sporadic task systems with explicit deadlines, the response time analysis technique has been developed. It is based on a scenario in which all tasks release jobs at the same time instant with all following jobs being released as early as permitted. This maximizes the response time \( R_i \) of the task in question, which is why the scenario is often called the critical instant. It is shown by Joseph and Pandya [28] and independently by Audsley et al. [5] that
$R_i$ is the smallest positive solution of the recurrence relation

$$R = E_i + \sum_{j < i} \left\lceil \frac{R}{P_j} \right\rceil \cdot E_j,$$

(9.4)

assuming that the tasks are in order of descending priority. This is based on the observation that the interference from a higher priority task $T_j$ to $T_i$ during a time interval of size $R$ can be computed by counting the number of jobs task $T_j$ can release as $\left\lceil \frac{R}{P_j} \right\rceil$ and multiplying that with their worst-case duration $E_j$. Together with $T_i$’s own WCET $E_i$, the response time is derived. Solving Equation (9.4) leads directly to a pseudo-polynomial schedulability test. Note that our response-time analysis for DRT task systems in Chapter 7 is fundamentally based on the same observation and a highly generalized variant. Eisenbrand and Rothvoß show in [19] that the problem of computing $R_i$ is indeed $\text{NP}$-hard.

### 9.2.2 The Multiframe Model

The first extension of the periodic and sporadic paradigm for jobs of different types to be generated from the same task was introduced by Mok and Chen in [35]. The motivation is as follows. Assume a workload which is fundamentally periodic but it is known that every $k$-th job of this task is extra long. As an example, Mok and Chen describe an MPEG video codec that uses different types of video frames. Video frames arrive periodically, but frames of large size and thus large decoding complexity are processed only once in a while. The sporadic task model would need to account for this in the WCET of all jobs, which is certainly a significant over-approximation. Systems that are clearly schedulable in practice would fail standard schedulability tests for the sporadic task model. Thus, in scenarios like this where most jobs are close to an average computation time which is significantly exceeded only in well-known periodically recurring situations, a more precise modeling formalism is needed.

To solve this problem, Mok and Chen introduce in [35] the Multiframe model. A Multiframe task $T$ is described as a pair $(P,E)$ much like the basic sporadic model with implicit deadlines, except that $E = (E_0, \ldots, E_{k-1})$ is a vector of different execution times, describing the WCET of $k$ potentially different frames. Note that a Multiframe task is a special case of a DRT task $T$ for which $G(T)$ is a directed cycle graph with $k$ vertices $V(T) = \{v_0, \ldots, v_{k-1}\}$ in which each vertex $v_i \in V(T)$ is labeled with $\langle e(v_i), d(v_i) \rangle = \langle E_i, P \rangle$ and each edge $(v_i, v_{i+1 \mod k}) \in E(T)$ with $p(v_i, v_{i+1 \mod k}) = P$.

### Semantics

As before, let $\rho = (J_0,J_1,\ldots)$ be a job sequence with job parameters $J_i = (R_i, e_i, v_i)$ of release time $R_i$, execution time $e_i$ and job type $v_i$. For $\rho$ to be gen-
generated by a Multiframe task $T$ with $k$ frames, it has to hold that $e_i \leq E_{(a+i)} \mod k$ for some offset $a$, i.e., the worst-case execution times cycle through the list specified by vector $E$. The job release times behave as before for sporadic implicit-deadline tasks, i.e., $R_{i+1} \geq R_i + P$. We show an example in Figure 9.2.

![Figure 9.2. Example of a Multiframe task $T = (P, E)$ with $P = 4$ and $E = (3, 1, 2, 1)$. Note that deadlines are implicit.](image)

**Schedulability Analysis**

Mok and Chen provide in [35] a schedulability analysis for static priority scheduling. They provide a generalization of Equation (9.3) by showing that a task set $\tau$ is schedulable with a static priority scheduler under rate-monotonic priority ordering if

$$U(\tau) \leq r \cdot n \cdot \left( (1 + 1/r)^{1/n} - 1 \right).$$

(9.5)

The value $r$ in this test is the minimal ratio between the largest WCET $E_i$ in a task and its successor $E_{(i+1)} \mod k$. Note that the classic test for periodic tasks in Equation (9.3) is a special case of (9.5) with $r = 1$.

The proof for this condition is done by carefully observing that for a class of Multiframe tasks called *accumulatively monotonic (AM)*, there is a critical instant that can be used to derive the condition (and further even for a precise test in pseudo-polynomial time by simulating the critical instant). In short, AM means that there is a frame in each task such that all sequences starting from this frame always have a cumulative execution demand at least as high as equally long sequences starting from any other frame. After showing (9.5) for AM tasks the authors prove that each task can be transformed into an AM task which is equivalent in terms of schedulability. The transformation is via a model called General Tasks from [34] which is an extension of Multiframe tasks to an infinite number of frames and therefore of mainly theoretical interest.

Refined sufficient tests have been developed [27, 13, 47, 32] with less pessimism than the test using the utilization bound in (9.5). They generally also allow certain task sets of higher utilization than those passing the above test to be classified as schedulable. A precise test of exponential complexity is presented in [51] based on response time analysis as a generalization of (9.4). The authors also include results for models with jitter and blocking.
9.2.3 Generalized Multiframe Tasks

In the Multiframe model, all frames still have the same period and implicit deadline. Baruah et al. generalize this further in [12] by introducing the *Generalized Multiframe (GMF)* task model. A GMF task $T = (P, E, D)$ with $k$ frames consists of three vectors:

$P = (P_0, \ldots, P_{k-1})$ for minimum inter-release separations,

$E = (E_0, \ldots, E_{k-1})$ for worst-case execution times, and

$D = (D_0, \ldots, D_{k-1})$ for relative deadlines.

For unambiguous notation we write $P^T_i$, $E^T_i$ and $D^T_i$ for components of these three vectors in situations where it is not clear from the context which task $T$ they belong to. Again, this model is easily seen to be another special case of the DRT task model. In particular, each GMF task $T$ can be expressed as a cycle graph $G(T)$. In contrast to the Multiframe model, all vertex and edge labels can be different from each other.

**Semantics**

As a generalization of the Multiframe model, each job $J_i = (R_i, e_i, v_i)$ in a job sequence $p = (J_0, J_1, \ldots)$ generated by a GMF task $T$ needs to correspond to a frame and the corresponding values in all three vectors. Specifically, we have for some offset $a$ that:

1. $R_{i+1} \geq R_i + P_{(a+i) \mod k}$
2. $e_i \leq E_{(a+i) \mod k}$

An example is shown in Figure 9.3.

![Figure 9.3. Example of a GMF task $T = (P, E, D)$ with $P = (5, 3, 4)$, $E = (3, 1, 2)$ and $D = (3, 2, 3)$.](image)

**Feasibility Analysis**

Baruah et al. give in [12] a feasibility analysis method based on the demand-bound function. The different frames make it difficult to develop a closed-form expression like (9.2) for sporadic tasks since there is in general no unique critical instant for GMF tasks. Instead, the described method (which we sketch here with slightly adjusted notation and terminology) creates a list of demand pairs $\langle e, d \rangle$, see Definition 3.2.1 on page 26. Recall that each demand pair $\langle e, d \rangle$ describes that a task $T$ can create $e$ time units of execution time demand during an interval of length $d$. From this information it can be derived that
\[ dbf_T(d) \geq e \] since the demand-bound function \( dbf_T(d) \) is the maximal execution demand possible during any interval of that size.

In order to derive all relevant demand pairs for a GMF task, Baruah et al. first introduce a property called localized Monotonic Absolute Deadlines (l-MAD) which we described in more detail in Section 2.2.1. Intuitively, it means that two jobs from the same task that have been released in some order will also have their (absolute) deadlines in the same order. Formally, this is can be expressed as

\[ D_i \leq P_i + D_{(i+1) \mod k}, \]

which is more general than the classical notion of constrained deadlines, i.e., \( D_i \leq P_i \), but still sufficient for the analysis. We assume this property for the rest of this section.

As preparation, the method from [12] creates a sorted list \( DP \) of demand pairs \( \langle e, d \rangle \) for all \( i \) and \( j \) each ranging from 0 to \( k - 1 \) with

\[
e = \sum_{m=i}^{i+j} E_m \mod k, \quad d = \left( \sum_{m=i}^{i+j-1} P_m \mod k \right) + D_{(i+j) \mod k}.
\]

(9.6)

For a particular pair of \( i \) and \( j \), this computes in \( e \) the accumulated execution time of a job sequence with jobs corresponding to frames \( i, \ldots, (i+j) \mod k \). The value of \( d \) is the time from first release to last deadline of such a job sequence. With all these created demand pairs, and using shorthand notation \( P_{\text{sum}} := \sum_{i=0}^{k-1} P_i \), \( E_{\text{sum}} := \sum_{i=0}^{k-1} E_i \) and \( D_{\text{min}} := \min_{i=0}^{k-1} D_i \), the function \( dbf_T(t) \) can be computed with

\[
dbf_T(t) = \begin{cases} 0 & \text{if } t < D_{\text{min}}, \\ \max \{ e \mid \langle e, d \rangle \in DP \text{ with } d \leq t \} & \text{if } t \in [D_{\text{min}}, P_{\text{sum}} + D_{\text{min}}), \\ \left\lfloor \frac{t - D_{\text{min}}}{P_{\text{sum}}} \right\rfloor E_{\text{sum}} + dbf_T(D_{\text{min}} + (t - D_{\text{min}}) \mod P_{\text{sum}}) & \text{if } t \geq P_{\text{sum}} + D_{\text{min}}. \end{cases}
\]

(9.7)

Intuitively, we can sketch all three cases as follows: In the first case, time interval \( t \) is shorter than the shortest deadline of any frame, thus not creating any demand. In the second case, time interval \( t \) is shorter than \( P_{\text{sum}} + D_{\text{min}} \) which implies that at most \( k \) jobs can contribute to \( dbf_T(t) \). All possible job sequences of up to \( k \) jobs are represented in demand pairs in \( DP \), so it suffices to return the maximal demand \( e \) recorded in a demand pair \( \langle e, d \rangle \) with \( d \leq t \).

In the third case, a job sequence leading to the maximal value \( dbf_T(t) \) must include at least one complete cycle of all frames in \( T \). Therefore, it is enough to determine the number of cycles (each contributing \( E_{\text{sum}} \)) and looking up the remaining interval part using the second case.

Finally, [12] describes how a feasibility test procedure can be implemented by checking Condition (9.1) for all \( t \) at which \( dbf(t) \) changes up to a bound

\[
D := \frac{U(\tau)}{1 - U(\tau)} \cdot \max_{T \in \tau} (P^T_{\text{sum}} - D^T_{\text{min}})
\]

with \( U(\tau) := \sum_{T \in \tau} E^T_{\text{sum}} / P^T_{\text{sum}} \) measuring the utilization of a GMF task system. If \( U(\tau) \) is bounded by a constant \( c < 1 \) then this results in a feasibility test of
pseudo-polynomial complexity. Baruah et al. include also an extension of this method to task systems without the \textit{l-MAD} property, i.e., with arbitrary deadlines. As an alternative test method, they even provide an elegant reduction of GMF feasibility to feasibility of sporadic task sets by using the set \textit{DP} to construct a \textit{dbf}-equivalent sporadic task set.

**Static Priorities**

An attempt to solve the schedulability problem for GMF in the case of static priorities was presented by Takada and Sakamura [45]. The idea is to use a function called \textit{Maximum-interference function (MIF) }$M(t)$. It is based on the \textit{request-bound function }$rbf(t)$ which for each interval size $t$ counts the accumulated execution demand of jobs that can be \textit{released} inside any interval of that size. (Notice that in contrast, the demand-bound function also requires the job deadline to be inside the interval.) The MIF is a “smoother” version of that, which for each task only accounts for the execution demand that could actually execute inside the interval. We show examples of both functions in Figure 9.4.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig9_4.png}
\caption{Examples of request-bound function and maximum-interference function of the same task.}
\end{figure}

The method uses the MIF as a generalization in the $\sum$-summation term in Equation (9.4), leading to a generalized recurrence relation for computing the response time:

$$R = E_i + \sum_{j<i} M_j(R)$$

(9.8)

Note that this expresses the response time of a \textit{job} generated by one particular \textit{frame} $i$ of a GMF task with $M_j(t)$ expressing the corresponding maximum-interference functions of higher priority tasks. Computation of $M_j(t)$ is essentially the same process as determining the demand-bound function $dbf(t)$ from above. This is a very similar approach as our response-time analysis procedure in Chapter 7. It roughly corresponds to the first iterative step of our refinement scheme where each task is represented with its most abstract request function (which is in fact equal to the request-bound function).
Our hardness result from Chapter 5 shows that the proposed method does not lead to a precise test since the response time computed by solving Equation (9.8) is over-approximate. The reason is that $M_i(t)$ over-approximates the actual interference caused by higher priority tasks. See Figure 6.8 on page 98 for an example which also applies to the MIF abstraction.

Generally, the hardness result from Chapter 5 establishes that this is inherent by demonstrating that one single integer-valued function on the time interval domain cannot adequately capture the information needed to compute exact response times. Different concrete task sets with different resulting response times need to be abstracted by identical functions, ruling out a precise test. Indeed, our result in Chapter 5 shows that the problem of an exact schedulability test for GMF tasks in case of static priority schedulers is strongly coNP-hard implying that there is no adequate replacement for $M_i(t)$. Recall that the rather involved proof is mostly focussing on the DRT task model but is shown to even hold in the GMF case. Still, the test for GMF presented in [45] is a sufficient test of pseudo-polynomial time complexity.

## 9.2.4 Non-cyclic GMF

The original motivation for Multiframe and GMF task models was systems consisting of frames with different computational demand and possibly different deadlines and inter-release separation times, arriving in a pre-defined pattern. Consider again the MPEG video codec example where video frames of different complexity arrive, leading to applicability of the Multiframe model. For the presented analysis methods, the assumption of a pre-defined release pattern is fundamental. Consider now a system where the pattern is not known a priori, for example if the video codec is more flexible and allows different types of video frames to appear adaptively, depending on the actual video contents. Similar situations arise in cases where the frame order depends on other environmental decisions, e.g. user input or sensor data. A prominent example is an engine management component in an automotive embedded real-time system. Depending on the engine speed, the tasks controlling ignition timing, fuel injection, opening of exhaust valves, etc. have different periods since the angular velocity of the crankshaft changes [18]. These systems cannot be modeled with the GMF task model.

Moyo et al. propose in [46] a model called Non-Cyclic GMF to capture such behavior adequately. A Non-Cyclic GMF task $T = (P, E, D)$ is syntactically identical to GMF task from Section 9.2.3, but with non-cyclic semantics. In order to define the semantics formally, let $\phi : \mathbb{N} \rightarrow \{0, \ldots, k - 1\}$ be a function choosing frame $\phi(i)$ for the $i$-th job of a job sequence. Having $\phi$, each job $J_i = (R_i, e_i, v_i)$ in a job sequence $\rho = (J_0, J_1, \ldots)$ generated by a non-cyclic GMF task $T$ needs to correspond to frame $\phi(i)$ and the corresponding values in all three vectors:
1. \( R_{i+1} \geq R_i + P_{\phi(i)} \)
2. \( e_i \leq E_{\phi(i)} \)

This contains cyclic GMF job sequences as the special case where \( \phi(i) = (a + i) \mod k \) for some offset \( a \). An example of non-cyclic GMF semantics is shown in Figure 9.5.

![Figure 9.5. Non-cyclic semantics of the GMF example Figure 9.3](image)

As for Multiframe and GMF models, the non-cyclic GMF model is a special case of the DRT task model in which \( G(T) \) for a task \( T \) is a complete digraph. This digraph has the special property that all edges from the same vertex have the same label. Figure 9.6 illustrates the different ways of representing a GMF task with both semantics as a DRT task.

\[
T = (P, E, D)
\]

\[
P = (10, 8, 3, 5, 5)
\]

\[
E = (1, 2, 3, 1, 1)
\]

\[
D = (10, 7, 7, 9, 8)
\]

![Figure 9.6. Different ways of representing a GMF task T](image)

For analyzing non-cyclic GMF models, Moyo et al. give in [46] a simple density-based sufficient feasibility test. Defining \( D(T) := \max_i C^T_i / D^T_i \) as the density of a task \( T \), a task set \( \tau \) is schedulable if \( \sum_{T \in \tau} D(T) \leq 1 \). This generalizes a similar test for the sporadic task model with explicit deadlines. In addition to this test, [46] also includes an exact feasibility test based on efficient systematic simulation.

A different exact feasibility test is presented by Baruah in [10] for constrained deadlines using the demand-bound function as in Condition (9.1). A dynamic programming approach is used to compute demand pairs (see Sec-
tion 9.2.3) based on the observation that $dbf_T(t)$ can be computed for larger and larger $t$ reusing earlier values. More specifically, a function $A_T(t)$ is defined which denotes for an interval size $t$ the accumulated execution demand of any job sequence where jobs have their full exclusion window\(^1\) inside the interval. It is shown that $A_T(t)$ for $t > 0$ can be computed by assuming that some frame $i$ was the last one in a job sequence contributing a value to $A_T(t)$. In that case, the function value for the remaining job sequence is added to the execution time of that specific frame $i$. Since frame $i$ is not known a priori, the computation has to take the maximum over all possibilities. Formally,

$$A_T(t) = \max_i \left\{ A_T(t - P^T_i) + E^T_i \mid P^T_i \leq t \right\}.$$  \hspace{1cm} (9.9)

Using this, $dbf_T(t)$ can be computed via the same approach by maximising over all possibilities of the last job in a sequence contributing to $dbf_T(t)$. It uses that the execution demand of the remaining job sequence is represented by function $A_T(t)$, leading to

$$dbf_T(t) = \max_i \left\{ A_T(t - D^T_i) + E^T_i \mid D^T_i \leq t \right\}.$$  \hspace{1cm} (9.10)

This leads to a pseudo-polynomial time bound for the feasibility test if $U(\tau)$ is bounded by a constant, since $dbf(t) > t$ implies $t < \left( \sum_{i} E^T_i \right) / (1 - U(\tau))$ which is pseudo-polynomial in this case. Note that both this bound and the method for computing $dbf_T$ are in fact special cases of our method for analyzing DRT feasibility in Chapter 3.

The same article also proves that evaluating the demand-bound function is a (weakly) $NP$-hard problem. More precisely: Given a non-cyclic GMF task $T$ and two integers $t$ and $B$ it is $coNP$-hard to determine whether $dbf_T(t) \leq B$. The proof is via a rather straightforward reduction from the Integer Knapsack problem. Thus, a polynomial algorithm for computing $dbf(t)$ is unlikely to exist.

**Static Priorities**

A recent result by Berten and Goossens [15] proposes a sufficient schedulability test for static priorities. It is based on the request-bound function similar to [45] and its efficient computation. Similar to the approach in [45] the function is inherently over-approximate and the test is of pseudo-polynomial time complexity.

Another recent result by Davis et al. [18] introduces another similar sufficient analysis with extensions to constrained timing for releases of different frames. More specifically, if there is a priori knowledge about the minimal time that one job type has to be repeatedly released until a new type can be

\(^1\)The exclusion window of a job is the time interval between its release time and the earliest possible release time of the next job from the same task. In the GMF model, this window has length $P^T_i$ for jobs released by frame $i$ of task $T$. 

140
instantiated, the authors give an improved analysis method. This extension is motivated by practically limited acceleration of engine speed in combustion engines from the automotive domain. This is another example of global timing constraints, cf. Chapter 4.

9.3 Graph-oriented Models

The more expressive workload models become, the more complicated structures are necessary to describe them. In this section we turn to models based on different classes of directed graphs. Recall that we already interpreted all task models from the previous section in terms of graph classes and thereby special cases of the DRT task model, cf. also Figure 9.6. The difference is that the models in this section were in fact already originally defined in terms of classes of directed graphs.

9.3.1 Recurring Branching Tasks

A first generalization to the GMF model was presented in [8], followed up by [3]. It is based on the observation that real-time code may include branches that influence the pattern in which jobs are released. As the result of some branch, a sequence of jobs may be released which may differ from the sequence released in a different branch. In a schedulability analysis, none of the branches may be universally worse than the others since that may depend on the situation, e.g., which tasks are being scheduled together with the branching one. Thus, all branches need to be modeled explicitly and a proper representation is needed, different from the GMF release structure.

A natural way of representing branching code is a tree. Indeed, the model proposed in [8] is a tree representing job releases and their minimum inter-release separation times. We show an example in Figure 9.7(a). Formally, a Recurring Branching (RB) task $T$ is a directed tree $G(T) = (V, E)$ in which, as for digraphs in DRT tasks, each vertex $v \in V$ represents a type of job to be released and each edge $(u, v) \in E$ the minimum inter-release separation times. They have labels $(e(v), d(v))$ and $p(u, v)$ as before. In addition to the tree, each leaf $u$ has a separation time $p(u, v_{\text{root}})$ to the root vertex $v_{\text{root}}$ in order to model that the behavior recurs after each traversal of the tree.

In order to simplify the feasibility analysis, the model is syntactically restricted in the following way. For each path $\pi = (\pi_0, \ldots, \pi_l)$ of length $l$ from the root $\pi_0 = v_{\text{root}}$ to a leaf $v_l$, its duration when going back to the root must be the same, i.e., the value $P := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + p(\pi_l, v_{\text{root}})$ must be independent of $\pi$. We call $P$ the period of $T$. Note that this is a generalization of GMF since GMF can be expressed as a linear tree.
Semantics

A job sequence $\rho = (J_0, J_1, \ldots)$ is generated by an RB task $T$ if it corresponds to a path $\pi$ through $G(T)$ in the following way. Path $\pi$ starts at some vertex $\pi_0$ in $G(T)$, follows the edges to a leaf, then starts again at the root vertex, traverses $G(T)$ again in a possibly different way, etc. (Very short $\pi$ may of course never reach a leaf.) The correspondence between $\rho$ and $\pi$ means that all jobs $J_i$ are of the form $J_i = (R_i, e_i, \pi_i)$, and the following:

1. $R_{i+1} \geq R_i + p(\pi_i, \pi_{i+1})$,
2. $e_i \leq e(\pi_i)$.

Note that this is clearly subsumed by DRT semantics, cf. Section 2.2.2.

Feasibility Analysis

The analysis presented in [8] is based on the concept of demand pairs as introduced in Chapter 3. We sketch the method from [8], slightly adjusted to fit our terminology and notation. First, a set $DP_0$ is created consisting of all demand pairs corresponding to paths not containing both a leaf and a following root vertex. This is straightforward since for each pair of vertices $(u, v)$ in $G(T)$ connected by a directed path $\pi$, this connecting path is unique. Thus, a demand pair $\langle e, d \rangle$ can be created by enumerating all vertex pairs $(u, v)$ and computing for their connecting path $\pi = (\pi_0, \ldots, \pi_l)$ the values

$$
e := \sum_{i=0}^{l} e(\pi_i), \quad d := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + d(\pi_l).$$

Second, all paths $\pi$ which do contain both a leaf and a following root vertex can be cut into three subpaths $\pi_{\text{head}}, \pi_{\text{middle}}$ and $\pi_{\text{tail}}$:

$$\pi = (v_1, \ldots, v_{\text{root}}, v'_{\text{root}}, v''_{\text{root}}, \ldots, v'_{\text{root}}, v'', \ldots, v'', \ldots, v''').$$

We use $v_1, v'_1, \text{etc.}$ for arbitrary leaf nodes. The first part $\pi_{\text{head}}$ is the prefix of $\pi$ up to and including the first leaf in $\pi$. The second part $\pi_{\text{middle}}$ is the middle
part starting with \(v_{\text{root}}\) and ending in the last leaf which \(\pi\) visits. Note that \(\pi_{\text{middle}}\) may traverse the tree several times. The third part \(\pi_{\text{tail}}\) starts with the last occurrence of \(v_{\text{root}}\) in \(\pi\). For each of the three parts, a data structure is created so that demand pairs for a full path \(\pi\) can be assembled easily.

In order to represent \(\pi_{\text{head}}\), a set \(UP_{\text{leaf}}\) is created. For all paths \(\pi_{\text{head}} = (\pi_0, \ldots, \pi_l)\) that end in a leaf it contains a pair \(\langle e, p \rangle\) with

\[
e := \sum_{i=0}^{l} e(\pi_i), \quad p := \sum_{i=0}^{l-1} p(\pi_i, \pi_{i+1}) + p(\pi_l, v_{\text{root}}).
\]

For representing \(\pi_{\text{middle}}\), the maximal accumulated execution demand \(e_{\text{max}}\) of any path completely traversing the tree is computed. Note that all paths from the root to a leaf have the same sum of inter-release separation times and this sum is the period \(P\) of \(T\). Finally, for representing \(\pi_{\text{tail}}\), a set \(DP_{\text{root}}\) is computed as a subset of \(DP_0\) only considering paths starting at \(v_{\text{root}}\).

Using these data structures, \(dbf_T(t)\) can be computed easily. If \(t \leq P\), then a job sequence contributing to \(dbf_T(t)\) either corresponds to a demand pair in \(DP_0\) (not passing \(v_{\text{root}}\)) or is represented by items from \(UP_{\text{leaf}}\) and \(DP_{\text{root}}\) (since it is passing \(v_{\text{root}}\) exactly once):

\[
F_1(t) := \max \left\{ e \mid \langle e, d \rangle \in DP_0 \text{ with } d \leq t \right\},
F_2(t) := \max \left\{ e_1 + e_2 \mid \langle e_1, p \rangle \in UP_{\text{leaf}} \land \langle e_2, d \rangle \in DP_{\text{root}} \land p + d \leq t \right\},
\]

\[
dbf_T(t) = \max \{ F_1(t), F_2(t) \} \quad \text{if } t \leq P. \tag{9.11}
\]

In case \(t > P\), such a job sequence must pass through \(v_{\text{root}}\) and traverses the tree completely for either \(\lfloor t/P \rfloor\) or \(\lfloor t/P \rfloor - 1\) times. For the parts that can be represented by \(\pi_{\text{head}}\) and \(\pi_{\text{tail}}\) of the corresponding path \(\pi\), we can use \(F_2\) from above, since \(\pi_{\text{head}}\) concatenated with \(\pi_{\text{tail}}\) correspond to a job sequence without a complete tree traversal but which visits \(v_{\text{root}}\) exactly once. Putting it together for \(t > P\):

\[
F_3(t) := \left\lfloor \frac{t}{P} \right\rfloor \cdot e_{\text{max}} + F_2(t \mod P), \tag{9.12}
F_4(t) := \left\lfloor \frac{t - P}{P} \right\rfloor \cdot e_{\text{max}} + F_2((t \mod P) + P), \tag{9.13}
\]

\[
dbf_T(t) = \max \{ F_3(t), F_4(t) \} \quad \text{if } t > P. \tag{9.14}
\]

Finally, in order to do the feasibility test, i.e., verify Condition (9.1), the demand-bound function \(dbf_\tau(t) = \sum_{T \in \tau} dbf_T(t)\) is computed for all \(t\) up to a bound \(D\) derived in a similar way as for GMF in Section 9.2.3.

### 9.3.2 Recurring Real-Time Tasks – DAG Structures

In typical branching code, the control flow is joined again after the branches are completed. Thus, no matter which branch is taken, the part after the join
is common to both choices. In the light of a tree release structure as in the RB task model above, this means that many vertices in the tree may actually represent the same types of jobs to be released, or even whole subtrees are equal. In order to make use of these redundancies, Baruah proposes in [7] to use a directed acyclic graph (DAG) instead of a tree. The impact is mostly efficiency: each DAG can be unrolled into a tree, but that comes at the cost of potentially exponential growth of the graph.

A Recurring Real-Time (RRT) task $T$ is a directed acyclic graph $G(T)$. The definition is very similar to RB tasks in the previous section, we only point out the differences. It is assumed that $G(T)$ contains one unique source vertex (corresponding to $v_{root}$ in an RB task) and further one unique sink vertex (corresponding to leafs in a tree). There is no explicit minimum inter-release separation time between the sink and the source vertex. Instead, an RRT task has an explicitly defined period parameter $P$ that constrains the minimum time between two releases of jobs represented by the source vertex. An RRT task behaves just like an RB task by following paths through the DAG. We skip the details and give an example of an RRT task in Figure 9.7(b).

**Feasibility Analysis**

Because of its close relation to RB tasks, the feasibility analysis method presented in [7] is very similar to the method presented above for RB tasks and we skip the details. Note that this is largely due to the general period parameter $P$ which could be inferred in the RB model since all paths from $v_{root}$ to $v_{root}$ were assumed to have the same duration. However, the adapted method has exponential complexity since it enumerates paths explicitly.

Chakraborty et al. present a more efficient method in [16] based on a dynamic programming approach, leading back to pseudo-polynomial complexity. Instead of enumerating all pairs of vertices $(u,v)$ in the DAG, the graph is traversed in a breadth-first manner. The critical observation is that all demand pairs representing a path ending in any particular vertex $v$ can be computed from those for paths ending in all parent vertices. It is not necessary to have precise information about which the actual paths are that the demand pairs represent. Even though Chakraborty et al. consider a limited variant of the model in which paths traverse the DAG only once, the ideas can be applied in general to the full RRT model. Note that this is an early special case of the iterative graph exploration approach presented in Chapter 3. It relies on the DAG structure, i.e., the assumption that there are no loops, which is not necessary for our method in Chapter 3. However, the general dynamic programming approach is based on a similar idea.

The feasibility problem is further shown to be $NP$-hard in [16] via a reduction from the Knapsack problem and the authors give a fully polynomial-time approximation scheme. For the special case where all vertices have equal WCET annotations, they show a polynomial-time solution, similar to the dynamic programming technique above.
Static Priorities

A sufficient test for schedulability of an RRT task set with static priorities is presented in [9]. It is shown that, up to a polynomial factor, the priority assignment problem in which a priority order has to be found is equivalent to the priority testing problem where a task set with a given priority order is to be tested for schedulability. At the core of both is the test whether a given task \( T \in \tau \) will meet its deadlines if it has the lowest priority of all tasks in \( \tau \) (whose relative priority order does not matter). In that case \( T \) is called lowest-priority feasible. Cf. our method in Chapter 6 which is based on the same framework scheme.

The proposed solution gives a condition involving both the demand-bound function \( \text{dbf}_T(t) \) and the request-bound function \( \text{rbf}_T(t) \). It is shown that a task \( T \) is lowest-priority feasible if

\[
\forall t \geq 0 : \exists t' \leq t : t' \geq \text{dbf}_T(t) + \sum_{T' \in \tau \setminus \{T\}} \text{rbf}_{T'}(t').
\]  

(9.15)

It is shown that \( \text{rbf}_T(t) \) can be computed with just a minor modification to the computation procedure of \( \text{dbf}_T(t) \) and that Condition (9.15) only needs to be checked for a bounded testing set of \( t \), similar to the bound \( D \) introduced in checking Condition (9.1) in feasibility tests. For each \( t \), checking the existence of a \( t' \leq t \) is essentially identical to an iterative procedure of solving the recurrence relation in Equation (9.8) of which (9.15) is a generalization.

A tighter and more efficient test is shown in [16] based on a smoother variant of the request-bound function, denoted \( \text{rbf}'_T(t) \). Using this, a task \( T \) is lowest-priority feasible if

\[
\forall v \in G(T) : \exists t' \leq d(v) : t' \geq e(v) + \sum_{T' \in \tau \setminus \{T\}} \text{rbf}'_{T'}(t').
\]  

(9.16)

This test is a more direct and tighter generalization of the sufficient test (9.8) for GMF tasks.

Position in the Model Hierarchy

A few comments about the position of the RRT model in the model hierarchy in Figure 9.1 on page 130 are in order.

First, the RRT model is a generalization of the RB model in the sense of Definition 9.1.3 on page 130. It is not the case that syntactically, every RB task is also an RRT task, because of the edges from leaves back to the root vertex in an RB task. However, an equivalent RRT task can be constructed as follows. Given \( G(T) \) of an RB task \( T \), we create a new vertex \( v \) serving as a dummy sink vertex, i.e., \( (e(v),d(v)) = (0,0) \). For each leaf vertex \( u \), we create an edge \((u,v)\) to the sink vertex with edge label \( p(u,v) := p(u,v_{\text{root}}) \) and remove edge \((u,v_{\text{root}})\). The resulting graph is \( G(T') \) of a new task \( T' \) which together with a parameter \( P \) equal to the duration of any path through
back to its root is now syntactically an RRT task. It is easily verified that $[T] \cong [T']$.

Second, the RRT model is, strictly speaking, not generalized by the DRT model. This is because the parameter $P$ for every RRT task is a global constraint which the DRT model cannot directly express. However, there is just one such constraint per RRT task. Thus, each RRT task is trivially generalized by a syntactically very similar 1-EDRT task, cf. Chapter 4. Recall that we show a task transformation technique from $k$-EDRT tasks to feasibility equivalent DRT tasks in Chapter 4. Therefore, the methods in this thesis for DRT and $k$-EDRT tasks practically generalize the above methods for RRT tasks.

9.3.3 Non-cyclic RRT

A different generalization of RB is non-cyclic RRT\(^2\) [11] where the assumption of one single sink vertex is removed. Specifically, a non-cyclic RRT task $T$ is a DAG $G(T)$ with vertex and edge labels as before that has a unique source vertex $v_{source}$. Additionally, for every sink vertex $v$, there is a value $p(v, v_{source})$ as before. We give an example in Figure 9.7(c). Note that a non-cyclic RRT task does not have a general period parameter, i.e., paths through $G(T)$ visiting $v_{source}$ repeatedly may do so in differing time intervals when doing so through different sinks.

Feasibility Analysis

The analysis technique presented by Baruah in [11] is similar to the ones of RB and RRT. The author uses the dynamic programming technique from [16] to compute demand pairs inside the DAG in order to keep pseudo-polynomial complexity and assumes a partition of paths $\pi$ into $\pi_{head}$, $\pi_{middle}$ and $\pi_{tail}$ as before. The difference here is that paths traversing $G(T)$ completely from $v_{source}$ to a sink may have different lengths, i.e., $\pi_{middle}$ is not necessarily a multiple of some period $P$. Thus, the expressions for partial dbf\(_T\)(\(t\)) computation in (9.12) and (9.13) can’t just assume a fixed length $P$ and a fixed computation time $e_{max}$. The idea to solve this is to first use the technique from [16] to compute demand pairs for full DAG traversals. These can then be interpreted as frames with a length and an execution time requirement, which can be concatenated to achieve a certain interval length, like a very big non-cyclic GMF task. Similar to (9.9) for solving non-cyclic GMF feasibility, all possible paths going from source to a sink can be represented in a function $f_{T}(t)$ that expresses for each $t$ the amount of execution demand these special paths may express.

\(^2\)The name “non-cyclic RRT” can be a bit misleading. The behavior of a non-cyclic RRT task is cyclic, in the sense that the source vertex is visited repeatedly. However, in comparison to the RRT model, the behavior is non-periodic, in the sense that revisits of the source vertex may happen in different time intervals.
create during intervals of length $t$. Similar to (9.10), this function is integrated into (9.12) and (9.13), resulting in an efficient procedure.

**Position in the Model Hierarchy**

Again, we give a short discussion regarding the position of the non-cyclic RRT model in the model hierarchy in Figure 9.1 on page 130.

First, it is clear that non-cyclic RRT is generalized by DRT since each non-cyclic RRT task is already syntactically a DRT task. In fact, the non-cyclic RRT model is the subclass of the DRT model where each $G(T)$ is a strongly connected directed graph in which *all* cycles share a common vertex.

Second, non-cyclic RRT generalizes non-cyclic GMF. This is not directly obvious since non-cyclic GMF does not syntactically fit into the non-cyclic RRT definition, even if interpreted as a (fully connected) digraph. However, a transformation of a non-cyclic GMF task $T = (P, E, D)$ can be given. In order to create $G(T')$ for an equivalent non-cyclic RRT task $T'$, we create $k$ vertices $v_0, \ldots, v_{k-1}$, each representing one of the $k$ frames of $T$, i.e., $\langle e(v_i), d(v_i) \rangle := (E_i, D_i)$. Further, we create a (dummy) source vertex $v_{source}$ with $\langle e(v_{source}), d(v_{source}) \rangle := (0, 0)$. We connect the source with the vertices $v_i$ via edges $(v_i, v_{source})$ and $(v_{source}, v_i)$ which have labels $p(v_i, v_{source}) := P_i$, $p(v_{source}, v_i) := 0$.

It is clear that $T'$ is equivalent to $T$ since all vertices $v_i$ can be visited in any order, just as the frames of $T$.

**9.4 Beyond DRT**

We now turn to models that either extend the DRT model in other directions or are outside the hierarchy in Figure 9.1 on page 130 since they operate partially or entirely on a different abstraction level.

**9.4.1 Task Automata**

A very expressive workload model called task automata is presented by Fersman et al. in [23]. It is based on Timed Automata that have been studied thoroughly in the context of formal verification of timed systems [1, 14]. Timed automata are finite automata extended with real-valued clocks to specify timing constraints as enabling conditions, i.e., guards on transitions. The essential idea of task automata is to use the timed language of an automaton to describe task release patterns.

In DRT terms, a task automaton (TA) $T$ is a graph $G(T)$ with vertex labels as in the DRT model, but labels on edges are more expressive. An edge $(u, v)$
is labeled with a guard $g(u,v)$ which is a boolean combination of clock comparisons of the form $x \diamondsuit C$ where $C$ is a natural number and $\diamondsuit \in \{\leq, <, \geq, >\}$. Further, an edge may be labeled with a clock reset $r(u,v)$ which is a set of clocks to be reset to 0 when this edge is taken. Since the value of clocks is an increasing real value which represents that time passes, guards and resets can be used to constrain timing behavior on generated job sequences. We give an example of a task automaton in Figure 9.8.

A task automaton has an initial vertex. In addition to resets and guards, task automata have a synchronization mechanism. This allows them to synchronize on edges either with each other or with the scheduler on a job finishing time.

![Figure 9.8. Example of a task automaton.](image)

Note that DRT tasks are special cases of task automata where only one clock is used. Each edge $(u, v)$ in a DRT task with label $p(u, v) = C$ can be expressed with an edge in a task automaton with guard $x \geq C$ and reset $x := 0$.

The authors of [21] show that the schedulability problem for a large class of systems modeled as task automata can be solved via a reduction to a model checking problem for ordinary Timed Automata. A tool for schedulability analysis of task automata is presented in [2, 22]. In fact, the feasibility problem is decidable for systems where at most two of the following conditions hold:

**Preemption.** The scheduling policy is preemptive.

**Variable Execution Time.** Jobs may execute for a time from an interval of possible execution times.

**Task Feedback.** The finishing time of a job may influence new job releases.

However, the feasibility problem is undecidable if all three conditions are true [21]. Task automata therefore mark a borderline between decidable and undecidable problems for workload models.

### 9.4.2 Real-Time Calculus

A more abstract formalism to model real-time workload is the **Real-Time Calculus (RTC)** introduced in [31]. Its abstractions are based on the notions of
Event streams, representing the workload,
Computing capacity, representing the computing resource for the workload,
Processing units, modeling the process of the workload being executed, using
up computing capacity.

Formally, a request function $R(t)$ models the accumulated amount of work-
load up to a time point $t$, and similarly, a capacity function $C(t)$ models the
amount of computation resource available until time $t$. Both functions are the
inputs to a processing unit, which outputs a new event stream modeled by a
function $R'(t)$, and leaves remaining computing capacity modeled by $C'(t)$.
For a Greedy Processing Component (GPC) [17], i.e., a processing unit which
processes incoming events as soon as possible, the four functions are related
by the following equations:

$$ R'(t) = \min_{0 \leq u \leq t} \{ R(u) + C(t) - C(u) \}, $$
$$ C'(t) = C(t) - R(t). $$

Modeling static priority scheduling is fairly straightforward in this model by
creating a number of processing units which all have their own input of re-
quest functions (representing different tasks) but all being chained through the
corresponding capacity functions, i.e., $C'(t)$ of one unit being $C(t)$ of the next
one. The later in this chain a task is processed, the lower its priority. Other
schedulers can also be modeled, with more involved constructions.

A concrete system may create many different request functions $R(t)$. In or-
der to represent these, Real-Time Calculus considers upper and lower bounds
in the time interval domain. Specifically, for abstracting the request function
$R(t)$, two arrival curves $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ are introduced, so that for all $t \geq 0$
and $\Delta \geq 0$

$$ \alpha^l(\Delta) \leq R(t + \Delta) - R(t) \leq \alpha^u(\Delta). $$

Clearly, a pair $(\alpha^u, \alpha^l)$ abstracts an infinite set of event streams. Analogously,
a pair of service curves $(\beta^u, \beta^l)$ is defined in the time domain to abstract com-
puting capacity.

All analysis can be executed on the level of arrival and service curves. For
example, the remaining service curve of a GPC can be computed via

$$ \beta^{lt}(\Delta) = \max_{0 \leq u \leq \Delta} \left\{ \beta^l(u) - \alpha^u(u) \right\}. $$

More generally, max-plus algebra can be used to elegantly describe the rela-
tions between arrival and service curves [17]. Response-time analysis can be
performed by deriving the response time from the horizontal distance between
$\alpha^u$ and $\beta^l$.

For many classes of systems, arrival curves can be specified directly with
closed form expressions. This includes the sporadic task model with an ex-
pression similar to (9.2) on page 132 and related models like periodic events with jitter [37].

It is worth noting that arrival curves and the request-bound function introduced earlier are equivalent concepts. However, RTC models systems are directly using arrival curves, i.e., they are the low-level construct used for system representation. In contrast, the request-bound function (and similarly, the demand-bound function) is a tool for schedulability analysis that abstracts from a more concrete system description. It is a potentially over-approximate abstraction and the process of deriving request- and demand-bound functions for task models presented in Sections 9.2 and 9.3 is more or less computationally demanding. Thus, analysis methods for RTC have the advantage that arrival curves can be assumed as given input to the analysis procedures, at the cost of introducing imprecision.

9.4.3 Fork-Join Real-Time Tasks

Recently, we proposed an extension of the DRT task model to incorporate fork/join structures. We sketch the overview from [40]. Instead of following just one path through the graph, the behavior of a task includes the possibility of forking into different paths at certain nodes, and joining these paths later. Syntactically, this is represented using hyperedges. A hypergraph generalizes the notion of a graph by extending the concept of an edge between two vertices to hyperedges between two sets of vertices.

More precisely, a hyperedge \((U, V)\) is either a sequence edge with \(U\) and \(V\) being singleton sets of vertices, or a fork edge with \(U\) being a singleton set, or a join edge with \(V\) being a singleton set. In all cases, the edges are labeled with a non-negative integer \(p(U, V)\) denoting the minimum job inter-release separation time. The model is illustrated with an example in Figure 9.9. Note that this contains the DRT model as a special case if all hyperedges are sequence edges.

As an extension of the DRT model, an FJRT task system releases independent jobs, allowing to define concepts like utilization \(U(\tau)\) and demand-bound function just as before in Chapter 3. A task executes by following a path through the hypergraph, triggering releases of associated jobs each time a vertex is visited. Whenever a fork edge \((\{u\}, \{v_1, \ldots, v_m\})\) is taken, \(m\) independent paths starting in \(v_1\) to \(v_m\), respectively, will be followed in parallel until joined by a corresponding join edge. In order for a join edge \((\{u_1, \ldots, u_n\}, \{v\})\) to be taken, all jobs associated with vertices \(u_1, \ldots, u_n\) must have been released and enough time must have passed to satisfy the join edge label. Forking can be nested, i.e., these \(m\) paths can lead to further fork edges before being joined. Note that meaningful models have to satisfy structural restrictions, e.g., each fork needs to be joined by a matching join, and control is not allowed to “jump” between parallel sections.
Figure 9.9. Example FJRT task. The fork edge is depicted with an intersecting double line, the join edge with an intersecting single line. All edges are annotated with minimum inter-release delays \( p(U,V) \). The vertex labels are omitted in this example. A possible job sequence containing jobs with their absolute release times and job types (but omitted execution times) is \( \sigma = [(0,v_1),(5,v_2),(5,v_5),(6,v_4),(7,v_3),(8,v_5),(16,v_6),(22,v_1)] \).

Feasibility

A complete method for analyzing FJRT task sets is not known at the time of writing. However, we sketch current approaches [40]. The usual demand-bound function based condition, i.e., checking \( \forall t \geq 0 : \sum_{T \in \tau} dbf_T(t) \leq t \), is applicable, since the Feasibility Theorem (cf. Theorem 3.1.4 on page 25) also holds for the FJRT task model.

Demand Tuples. For an FJRT task \( T \) without fork and join edges, \( dbf_T(t) \) can be evaluated by traversing its graph \( G(T) \) using a demand tuples abstraction as introduced in Section 3.2. We can extend this method to the new hyperedges by a recursive approach. Starting with “innermost” fork/join parts of the hypergraph, the tuples are merged at the hyperedges and then used as path abstractions as before. It can be shown that this method is efficient.

Utilization. Recall that just computing \( dbf_T(t) \) does not suffice for a finite test since it is also necessary to know which interval sizes \( t \) need to be checked. As for the DRT model, a bound can be derived from the utilization \( U(\tau) \) of a task set \( \tau \). However, it turns out that an efficient way of computing \( U(\tau) \) is surprisingly difficult to find. The difficulty comes from parallel sections in the task with loops of different periods which, when joined, exhibit the worst-case behavior in very long time intervals of not necessarily polynomial length.
This thesis has introduced a new graph-based task model for describing real-time workload. The trade-off between expressiveness and analysis efficiency has been studied using this model, which led to efficient analysis methods and hardness results. In particular, tractability borderlines have been established for the schedulability problem with earliest deadline first and static priority schedulers.

A classification of task models into tractable and intractable classes is shown in Figure 10.1. This provides interesting insights about the precise position of the borderlines which we will discuss briefly.

For EDF scheduling, the fundamental problem is to determine processor demand within time intervals. The complexity of computing this demand appears to be closely related to non-deterministic branching in task graphs. In the basic DRT model, cf. the hierarchy in Figure 10.1(a), computing demand is possible in pseudo-polynomial time as long as timing constraints are either purely local or can be translated into local constraints. In this case, graph traversal algorithms can track constraints one by one when following branches.
The extension from \(k\)-EDRT to EDRT causes intractability since many global constraints need to be considered together in all branches.

This phenomenon leads to a second tractability borderline for a hierarchy in which all models are equipped with global timing constraints, cf. Figure 10.1(b). The schedulability problem is intractable as long as models allow non-deterministic branching. Only the removal of branching possibilities allows pseudo-polynomial algorithms, since future extensions of paths develop deterministically.

For SP scheduling, the source of intractability is different. In this case, actual interference patterns need to be considered, in contrast to EDF. Different interference patterns can be combined in different ways, leading to a combinatorial explosion and therefore a high complexity of the schedulability problem. Therefore, all models allowing different types of frames can be shown to lead to intractability, cf. Figure 10.1(c). The non-cyclic GMF model using a complete digraph for each task plays a special role in the hierarchy. Even though it allows different types of frames, it can not enforce their order. Therefore, the hardness proofs in this thesis for static priorities are not applicable to the non-cyclic GMF model. Moreover, exact pseudo-polynomial time analysis methods are not known either and left open for future work.

Another direction for future work is the augmentation of the DRT task model with features like semaphores, shared resources or synchronisation. It is likely that combinatorial abstraction refinement is applicable to several of these problems as well, but this has not been investigated at the time of writing. A step in another direction is a fork-join based task model briefly described in Section 9.4.3, for which the complexity of analyzing feasibility is still an open problem.

Finally, an automata-theoretic view on the DRT task model may open new perspectives as well. A recent line of research [48] is linking automata and scheduling theory. The similarities of DRT tasks to finite automata may be investigated, for example by interpreting request functions as regular languages and thereby applying results from automata theory and the theory of formal languages to schedulability analysis.
The first person to be addressed in the Acknowledgements section of a thesis is usually the supervisor and this thesis will not diverge from this custom. Not because of tradition, but because he deserves it. Thanks to Wang Yi, my work at Polacksbacken was free from many worries that other Ph.D. students may encounter. He gave advice and most valuable feedback whenever needed, encouragement and trust, was always accessible and allowed an almost unbelievable level of freedom in my research. I am really grateful for that.

My research benefited greatly from close collaboration with my brilliant corridor mates Pontus Ekberg and Nan Guan and their fruitful insights and feedback. The resulting atmosphere of knowing that one can always go and connect to competent colleagues cannot be valued higher. Contributing to a positive work atmosphere were also the other Ph.D. students and senior researchers from “the second floor” (physically or mentally), creating fun memories not just at work.

I would like to thank Håkan Lanshammar, Michael Thuné and Ivan Cristoff as heads of department and division that manage to stay hidden in the background until a situation mandates their interference. It’s good to know that in case a problem occurs, several layers of hierarchy will be there to help you.

Some consider it a myth, but Ph.D. student life is not just about work. Thanks go to my friends from the Tuesday cards playing group as well as the Uppsala Academic Rowing Society (UARS) for making the years in Uppsala such a fun time.

I thank Pavel Krčál and my brother Roland Stigge for critical proof-reading of this thesis and Pontus Ekberg and Kai Lampka for language support regarding the Swedish and German summaries, respectively.

Finally, I thank my family for constant encouragement, love and support.
This thesis is based on the following six articles.


   I suggested the model, did most of the proofs, implemented a prototype and wrote the paper. The material from this article is the basis of Chapters 2 and 3.


   I suggested the model, did all proofs, implemented a prototype and wrote the paper. The material from this article is the basis of Chapter 4.


   I suggested the problem, did all proofs and wrote the paper. The material from this article is the basis of Chapter 5.


   I investigated all source material and wrote most of the paper. The material from this article is the basis of Chapter 9.


   I developed the algorithm, implemented a prototype, conducted all experiments and wrote the paper. The material from this article is the basis of Chapter 6 and contributes to Chapter 8.
6. **Refinement-based Exact Response-Time Analysis.** Martin Stigge, Nan Guan and Wang Yi. In submission.

I developed the overall algorithm and static priority characterization, implemented a prototype, conducted all experiments and wrote the paper. The material from this article is the basis of Chapter 7 and contributes to Chapter 8.

Additional Material

Some material in this thesis was not included in the original articles because of space or time reasons. The following list contains the major additional parts.

- Formulation and proof the Feasibility Theorem, i.e., Theorem 3.1.4.
- Formulation and proof of the cycle characterization of utilization, i.e., Lemma 3.2.9. This characterization was used as the original definition of utilization [39] but is now provided as a lemma because of a more general definition of utilization as asymptotic workload.
- Description of a very effective computation order optimization in Section 3.2.5 for DRT feasibility analysis.
- Hardness proofs for E-RB and E-ncGMF models, i.e., Theorem 4.4.2.
- Pseudo-polynomial feasibility analysis method for E-GMF presented in Section 4.4.2 including an efficient way of computing the utilization of E-GMF models.
- Extension of the hardness proof for static priority schedulability from the GMF to the multiframe model in Section 5.3.1.
- General formulation of the Combinatorial Abstraction Refinement procedure which is independent of the concrete schedulability problem, Section 6.4.
- Precise and formal definition of model generalization in Section 9.1, together with a model hierarchy derived from this definition and generalization details of RRT (Section 9.3.2) and non-cyclic RRT (Section 9.3.3).

Other Publications

The following list includes articles that I have coauthored but which are not included in this thesis.


För att kunna hantera sådana krav finns vanligen strikta tidsramar uppsatta inom specifikationer för inbyggda realtidssystem med säkerhetskritiska användningsområden. Om tidsramarna kan följas under systemets körning beror på flera faktorer, en av dem är exejveringstiden av de olika beräkningsenheter som systemet måste utföra. En annan viktig faktor är den ordning i vilken de olika beräkningsenhetera exekveras, vilket bestäms av en mjukvarukomponent som kallas schemaläggaren. Schemaläggaren ansvarar för att vid varje tidpunkt bestämma vilken av de beräkningsenheter som väntar på bli färdiga som ska exekveras först. En design av ett säkerhetskritiskt realtidssystem måste utsättas för grundlig validering för att garantera att allt kritiska tidsramar kommer att följas när systemet används. Det inkluderar en noggrann analys av schemaläggarens beteende. En matematiskt rigorös teknik är att utföra en så kallad schemaläggningsbarhetsanalys för att formellt verifiera en modell av beräkningsenhetera tillsammans med en modell av schemaläggaren. En sådan analys använder algoritmer för att matematiskt bevisa att inget av systemets beteenden kan bryta mot de givna tidsramarna.

Olika sätt att modellera beräkningsenheter har olika uttryckskraft och kan beskriva enheternas aktiveringsmönster på varierande detaljnivå, från traditionella periodiska modeller till sofistikerade grafbaserade modeller. En inneboende motsättning finns mellan uttryckskraften hos modelleringsformalismen och effektiviteten av schemaläggningsbarhetsanalysen. Ju mer uttryck-
skraft som finns i modelleringen, desto mer precis kan den beskriva beräkningsenhheterna, vilket reducerar överapproximation och därmed minimerar onödig överdimensionering av systemets resurser. Större uttrycks- kraft i modelleringen leder dock till högre komplexitet hos motsvarande analysmetod. En konsekvens av det är att en ideal modell har största möjliga uttrycks- kraft för vilken en effektiv och exakt analysmetod finns.

Zusammenfassung auf Deutsch


Verschiedene Modellformalismen erlauben die Beschreibung von Task-Aktivierungen mit unterschiedlicher Ausdrucksstärke, von traditionellen periodischen bis zu ausgefeilten Graph-basierten Formalismen. Ein inhärenter
Konflikt entsteht zwischen der Ausdrucksstärke von Modellen der Effizienz ihrer Analyse. Je ausdrucksstärker ein Modell ist, desto akkurater kann es die Rechenlast eines Systems beschreiben, was Überapproximation reduziert und damit unnötige Überdimensionierung von Systemressourcen minimiert. Jedoch führt höhere Ausdrucksstärke auch zu höherer Berechnungskomplexität der zugehörigen Analyseverfahren. Ein ideales Modell stellt also die maximale Ausdrucksstärke zur Verfügung, für welche noch effiziente Analysemethoden existieren.

Index

3-PARTITION problem, 69

Abstraction
  refinement, 98–99
  tree, 99

Acceptance ratio, 128

Branching task model, 68

Critical
  path, 40–42
  request function, 94–95
  vertex, 90–91

Deadline
  Arbitrary, 18
  Constrained, 17
  Implicit, 17
  Monotonic absolute, 17

Demand
  function, 112
  pair, 26
  quadruple, 44
  triple, 28

Demand function
  Suffix, 116

Demand-bound function, 23, 131

Domination
  of demand triples, 41
  of vertices, 90
  on request functions, 94

DRT model, 16–19

E-GMF model, 63

E-ncGMF model, 63

E-RB model, 63

EDF scheduler, 20

EDRT model, 47

Feasibility, 20
  Lowest priority, 87

Feasibility Theorem, 25

FJRT model, 150

Fork-join tasks, see FJRT model

GMF model, 63, 68, 135
  non-cyclic, 63, 138

Hamiltonian path problem, 60

Job, 15
  sequence, 18

Karp reduction, 59

l-MAD, 17

Lowest-priority feasibility, 87

Maximum-interference function, 137

Model hierarchy, 129

Multiframe model, 133

Non-cyclic GMF model, 63, 138

Non-cyclic RRT model, 146

Path
  Critical, 40–42
  Marked, 43
  Timed, 48

Periodic task model, 131

Pseudo-polynomial time, 59

RB model, 63, 141

Real-time calculus, 148

Request function, 92

Request-bound function, 137

Response time, 105

RRT model, 143
  non-cyclic, 146

Schedulability, 20

Scheduling window, 15

Semantics
  DRT, 18
  EDRT, 48
SP scheduler, 20
Sporadic task model, 131
Static priorities, see SP scheduler
Strongly \textit{NP}-hard, 60
Syntax
  \begin{itemize}
  \item DRT, 16
  \item EDRT, 47
  \end{itemize}

Task automata, 147

Utilization, 32

WCET, 10, 16
WCRT, 105
Workload function, 113
References


[47] Tei wei Kuo, Li pin Chang, Yu hua Liu, and Kwei jay Lin. Efficient online


