Diss. ETH No. 19648

Modular Performance Analysis of Embedded Real-Time Systems: Improving Modeling Scope and Accuracy

A dissertation submitted to

ETH ZURICH

for the degree of

Doctor of Sciences

presented by

SIMON PERATHONER
Dott. Ing. Inf. Politecnico Milano

born 12.06.1981
citizen of Italy

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm, co-examiner

2011

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 127

SIMON PERATHONER

Modular Performance Analysis of Embedded Real-Time Systems: Improving Modeling Scope and Accuracy

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A dissertation submitted to
ETH Zurich
for the degree of Doctor of Sciences

Diss. ETH No. 19648

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm, co-examiner

Examination date: April 21, 2011

Abstract

A fundamental aspect of the design of an embedded system is the prediction of its performance in terms of timing, memory, or energy early in the design process. The objective of this task, typically referred to as system-level performance evaluation, is twofold. On one hand, it is instrumental for pre-validating a system design before any resources are invested for the actual implementation and, on the other hand, the performance evaluation is a central driver for the exploration of the design space. For systems with strict performance requirements such as hard real-time systems the performance evaluation needs to be provably correct, that is, it has to cover the worst-case performance scenarios. Furthermore, the evaluation should be fast such that it can be employed for the exploration of large design spaces.

Recent research efforts have led to analytical and modular methods for worst-case performance evaluation at the system level. These methods ensure the correctness of the performance evaluation and are fast even for large-scale systems. However, they suffer from limited modelling scope and analysis accuracy. As a consequence, when applying these methods to complex systems, one often experiences considerable abstraction losses, which lead to overly pessimistic performance results.

This thesis introduces several formal models and methods that refine the modelling capabilities of analytical performance evaluation and prevent abstraction losses. The results build on the existing framework for Modular Performance Analysis (MPA), but apply also to other analytical formalisms. The main contributions of this thesis can be summarized as follows:

- The modelling scope of analytical performance evaluation is extended to systems with cyclic dependencies.
- New models and methods are introduced for handling structured event or data streams in analytical performance evaluation.
- A novel hybrid analysis methodology is presented that combines analytical and state-based system evaluation.
- New design methods for energy-efficient real-time systems are introduced.

Zusammenfassung

Ein wesentlicher Aspekt der Entwicklung von eingebetteten Systemen ist die Vorhersage deren Leistung bezüglich Zeitverhalten, Speicherbedarf oder Energieverbrauch in frühen Entwurfsphasen. Diese Leistungsbewertung, oft als *System-level Performance Evaluation* bezeichnet, dient zwei Zielen. Zum einen ermöglicht sie die Validierung von Systementwürfen vor der eigentlichen Implementierung des Systems. Zum anderen ist die Leistungsbewertung ein entscheidendes Element der Exploration des Entwurfsraumes. Für Systeme mit strengen Leistungsanforderungen wie zum Beispiel Echtzeitsystemen, muss die Leistungsbewertung nachweisbar korrekt sein, d.h. sie muss Worst-Case Szenarien erfassen. Weiterhin sollte die Leistungsbewertung möglichst schnell sein, damit sie für die Exploration von grossen Entwurfsräumen eingesetzt werden kann.

Die Forschungsbemühungen der letzten Jahre haben analytische, modulare Methoden zur Worst-Case Leistungsbewertung auf Systemebene hervorgebracht. Diese Methoden garantieren die Korrektheit der Leistungsbewertung und sind auch für grosse Systeme schnell. Allerdings zeichnen sich diese Verfahren auch durch einen eingeschränkten Anwendungsbereich und teilweise mangelnder Analysegenauigkeit aus. Aus diesem Grund kommt es bei der Analyse von komplexen Systemen häufig zu erheblichen Abstraktionsverlusten und folglich zu pessimistischen Leistungsbewertungen.

Die vorliegende Arbeit stellt formale Modelle und Methoden vor, die die Ausdruckskraft und Genauigkeit der analytischen Leistungsbewertung wesentlich verbessern. Die Arbeit baut auf den bestehenden Ansatz der *Modular Performance Analysis (MPA)* auf. Die Ergebnisse finden jedoch auch in weiteren analytischen Verfahren Anwendung. Die konkreten Beiträge dieser Arbeit können wie folgt zusammengefasst werden:

- Der Anwendungsbereich der analytischen Leistungsbewertung wird um Systeme mit zyklischen Abhängigkeiten erweitert.
- Modelle und Methoden zur Handhabung von strukturierten Datenströmen werden vorgestellt.
- Ein neuartiges hybrides Analyseverfahren wird eingeführt, welches analytische und zustandsbasierte Leistungsbewertung kombiniert.
- Neue Entwicklungsmethoden zum Entwurf von energieeffizienten Echtzeitsystemen werden vorgeschlagen.

Acknowledgement

First and foremost I would like to thank my advisor Prof. Dr. Lothar Thiele for his constant support during my research activity. Thank you for many fruitful discussions, and for all the valuable inputs that shaped this thesis. I would also like to thank Prof. Dr. Reinhard Wilhelm for co-examining this work.

Furthermore, I would like to express my sincere gratitude to all the collaborators that directly or indirectly contributed to the results of this thesis. In particular, I wish to thank Prof. Dr. Bengt Jonsson, Prof. Dr. Jian-Jia Chen, and Dr. Kai Lampka for sharing ideas and co-authoring papers.

I also thank all my current and former colleagues at the Computer Engineering group for their company and for lots of interesting discussions.

Finally, my warmest thanks go to my girlfriend Kathrin, my parents, and my brother for their constant love and support.

The work presented in this thesis was supported by the Swiss National Science Foundation (SNF) under grant number 200020-116594 an by the European Union project COMBEST under grant number 215543. This support is gratefully acknowledged.

Contents

Abstract i

Zusammenfassung iii

Acknowledgement v

1 Introduction 1
- 1.1 Embedded Real-Time Systems 1
- 1.2 Sources of Complexity 2
- 1.3 System-Level Performance Evaluation 4
  - 1.3.1 Role in the Design Flow 5
  - 1.3.2 Requirements 6
  - 1.3.3 Approaches 7
- 1.4 Thesis Overview and Contributions 9

2 Formal Methods for Performance Evaluation 13
- 2.1 Overview of related work 14
- 2.2 Modular Performance Analysis 19
  - 2.2.1 A general performance model 20
  - 2.2.2 Performance characterization in the time domain 22
  - 2.2.3 Real-Time Calculus 27
- 2.3 Evaluation and Comparison of Abstractions for Performance Verification 42
  - 2.3.1 Motivation and Challenges 42
  - 2.3.2 Comparison Methodology 43
  - 2.3.3 Benchmark Systems 44
  - 2.3.4 Analysis Results 47
  - 2.3.5 Discussion 56
- 2.4 Summary 58

3 Cyclic Dependencies 61
- 3.1 Introduction 62
- 3.2 Motivational example 63
- 3.3 Fixpoint Computations on Streams 65

3.3.1 Operational Model 65
3.3.2 Correctness and Convergence 70
3.4 Fixpoint Computations in MPA 74
3.4.1 Verification of Assumptions 74
3.4.2 Obtaining an Initial Characterization 77
3.4.3 Summary of the Method 77
3.5 Experiments 78
3.6 Related Work 81
3.7 Summary 83

4 Structured Event Streams 85
4.1 Introduction 85
4.1.1 Motivational example 86
4.1.2 Terminology 88
4.1.3 Organization 89
4.2 FIFO Scheduling 89
4.3 Event Count Curves 90
4.3.1 Definitions 91
4.3.2 Join and Fork of Simple Event Streams 93
4.3.3 Hierarchical application of ECCs 95
4.3.4 Join and Fork of Structured Event Streams 96
4.3.5 Experimental Evaluation 99
4.4 Case Study 101
4.4.1 General System Description 101
4.4.2 Detailed specification 102
4.4.3 Models and Analysis 103
4.4.4 Results 104
4.5 Related work 107
4.6 Summary 108

5 Hybrid Performance Verification 111
5.1 Introduction 112
5.1.1 Organization 113
5.2 Timed Automata 113
5.3 General Approach and Requirements 116
5.4 Interface MPA-TA 118
5.4.1 Linear Pattern 119
5.4.2 Concave/Convex Pattern 122
5.4.3 Extensions 129
5.4.4 Limiting the Complexity 131
5.5 Interface TA-MPA 132
5.5.1 Observer TA 133
5.5.2 Algorithm 136

5.5.3 Correctness 140
5.6 Experimental Evaluation 140
5.6.1 Case Study 141
5.6.2 Scalability of the Approach 145
5.6.3 Approximation Errors 150
5.7 Related work 153
5.8 Summary 155

6 Energy-Efficient System Design with MPA 157
6.1 Introduction 158
6.1.1 Organization 159
6.2 Dynamic Voltage Scaling 160
6.2.1 Offline DVS scheduling 161
6.2.2 Online DVS scheduling 161
6.3 Design of Offline DVS Systems with MPA 163
6.3.1 Models and Problem Definition 164
6.3.2 Proposed Algorithms 168
6.3.3 Experimental Evaluation 178
6.4 Design of Online DVS Systems with TA 182
6.4.1 System Model and Problem Definition 183
6.4.2 Adaptive Scheduling Scheme 183
6.4.3 Parametrization and Verification 186
6.4.4 Experimental Evaluation 192
6.5 Related Work 194
6.6 Summary 196

7 Conclusions 199
7.1 Main Results 199
7.2 Outlook 200

Bibliography 203

A Acronyms 217

List of Publications 219

Curriculum Vitae 221

1

Introduction

This thesis presents a set of novel formal methods for the performance evaluation of distributed embedded systems. It builds on an existing analytical framework for modular performance analysis, and aims at improving its modeling capabilities as well as the quality of the achievable analysis results. In this work we primarily consider real-time systems and, hence, focus on techniques for faithful timing analysis of systems. Existing analytical approaches for performance analysis are restricted to specific system models and performance metrics, and deliver only pessimistic performance guarantees for many real systems. The goal of our work is to extend the applicability of formal performance analysis methods by alleviating these problems.

The Introduction is structured as follows: In Section 1.1 we describe the main features of embedded real-time systems. The principal characteristics that complicate the design and the analysis of such systems are summarized in Section 1.2. Section 1.3 introduces the reader to system-level performance evaluation, describes its role in the design flow, and identifies its principal requirements. Finally, in Section 1.4 we give an overview of the thesis contents, and summarize our contributions.

1.1 Embedded Real-Time Systems

Embedded systems are computer systems that are dedicated to particular applications and are part of a larger device, often including electrical or mechanical parts. Examples of embedded systems are the hardware/software systems that control cars, airplanes, medical equipment, industrial robots, telecommunication devices, appliances, or consumer electronics. Nowadays, embedded processors account for by far the largest share of produced CPUs. This is not surprising if one considers, for instance, the rapidly increasing market for mobile electronic devices.

Embedded systems have several characteristics that distinguish them from general purpose computer systems. First of all, they are tightly coupled to the physical system they are embedded in, meaning that they continuously interact with it and have to meet constraints imposed by it. Note that in this aspect, there is a close relation between control systems and embedded systems: from a broad perspective, basically every computer-based controller in a control loop can be seen as an embedded system.

Another characteristic of embedded systems is that they are usually not reprogrammable by the end user of the device that embodies them. They are designed to perform just a few dedicated tasks that are known at design time. Knowledge about the particular system environment is often exploited for the design of optimized hardware/software solutions.

Moreover, in most of the cases embedded systems have to meet particular requirements in terms of energy, size, weight or cost. In many application domains, they also have to be fully predictable and highly dependable, as a malfunction or breakdown of the device they control is not acceptable.

A particular class of embedded systems are real-time systems. These are systems for which the execution has to meet timing constraints. For instance, in many cases an embedded system has to react to a stimulus or event emitted by the environment within a specified amount of time, also denoted as deadline. Note that real-time processing does not necessarily mean ’fast’ processing, but is a synonym for ’timely’ or ’predictable’ execution. Real-time systems for which a deadline violation cannot be tolerated are often denoted as *hard*.

Finally, we would like to note that recently many authors prefer the denomination *cyber-physical system* to *embedded system*. In particular, the expression *cyber-physical system* is often used to refer to networks of individual components which interact with their physical environment. Within the scope of this thesis, we do, however, not distinguish the two terms. For the sake of consistency, we will stick to the classical denomination of *embedded systems*.

1.2 Sources of Complexity

In many cases, modern embedded computer architectures are highly complex and consequently difficult to design and analyze. In the following we identify different factors that contribute to the complexity of embedded hardware/software systems.

- ***Distributed Architectures***
  Requirements of the controlled device such as scalability, fault tolerance or parallel completion of different (real-time) tasks very often lead to the design of distributed embedded systems. Furthermore, also the constraints imposed by the physical environment often enforce distributed solutions. A distributed system consists of a set of processing elements that communicate over some network. The distribution of the components considerably complicates the design and the analysis of a system. For instance, communication delays on shared busses are typically not negligible, and need to be considered in the timing analysis. Therefore, the analysis requires a holistic approach that considers both computation and communication in a distributed system.

- ***Heterogeneous Components***
  Another obstacle for the design and analysis of embedded systems is the fact that they often consist of heterogeneous components. The irregularity of the components very often derives from differing functionalities, or from particular characteristics of the local environments. For example, it is not unusual to find heterogeneous processing elements such as DSPs, microcontrollers, and (multi-core) CPUs in one and the same device. Similarly, the communication networks are often composed by multiple heterogeneous sub-networks.

- ***Multitasking***
  Frequently, the individual processing components of an embedded system execute multiple concurrent tasks. These processors implement a scheduling policy that determines which task is to be executed at which time. There are a variety of static and dynamic, preemptive and non-preemptive (real-time) scheduling policies. Depending on the particular scheduling policy and the activation pattern of the tasks, it is often not trivial to bound the timing interference among concurrent tasks.

- ***Shared Resources***
  Not only processors but also several other hardware resources are subject to contention in an embedded system. Examples are memories, communication channels, and I/O devices. Often the access to shared resources has to occur in a mutually exclusive way. Hence, while a task is granted access to a common resource, other tasks on

other processing elements might experience blocking, that is, they might be forced to wait before they gain access to the resource. Predicting the worst-case interference of tasks on each other is often hard, as the individual processing components operate in parallel, and usually take independent resource access decisions.

Note that all the concepts mentioned above can apply at different abstraction levels in the system architecture. For instance, the concept of networked components may apply to physically separated processors, but as well to Multiprocessor Systems-on-Chip (MPSoC). Similarly, contention for shared resources such as memories or busses can be observed on-chip as well as on a system-wide scale.

1.3 System-Level Performance Evaluation

Given the stringent requirements of many application domains, it becomes clear that evaluating the performance of an embedded system is highly important in the design process. In particular, not only the functional correctness of the computations performed by a system is relevant, but also the performance of the system in terms of e.g. response times or energy consumption needs to be ensured. Note that for embedded systems the distinction between functional (behavioural) and non-functional (performance) requirements is blurry. For instance, in a real-time system, a correct result arriving later (or even earlier) than specified can lead to a system failure. Similarly, in a battery operated mobile device, excessive energy consumption can compromise the required operation time of the device and hence its functionality.

Different performance aspects can be relevant for an embedded hardware/software system. The most common ones are:

- Timing properties (e.g. response times, end-to-end latencies)
- Memory requirements (e.g. buffer sizes)
- Energy consumption

The quantification of the system behaviour with respect to one or several of such metrics is generally referred to as performance evaluation or performance analysis.

The performance evaluation of embedded systems is essentially different from the performance evaluation of general-purpose computers. In particular, due to the restrictive requirements of many application domains (e.g. hard real-time constraints), the designers of embedded systems often can not rely on average or stochastic performance characterizations of a system. Instead, safe bounds for the best-case and worst-case performance of a system are required. This thesis focuses on high-level techniques for reliable worst-case (best-case) performance evaluation.

1.3.1 Role in the Design Flow

Performance evaluation at an early design stage, mainly at the system level, is important for taking fundamental design decisions before resources are invested in detailed implementations. In particular, the role of performance evaluation in the design flow is twofold: On one hand, it is employed as a validation or even certification instrument after the system-level design is completed. On the other hand, it is used as a driver for the design space exploration, as illustrated in Figure 1.

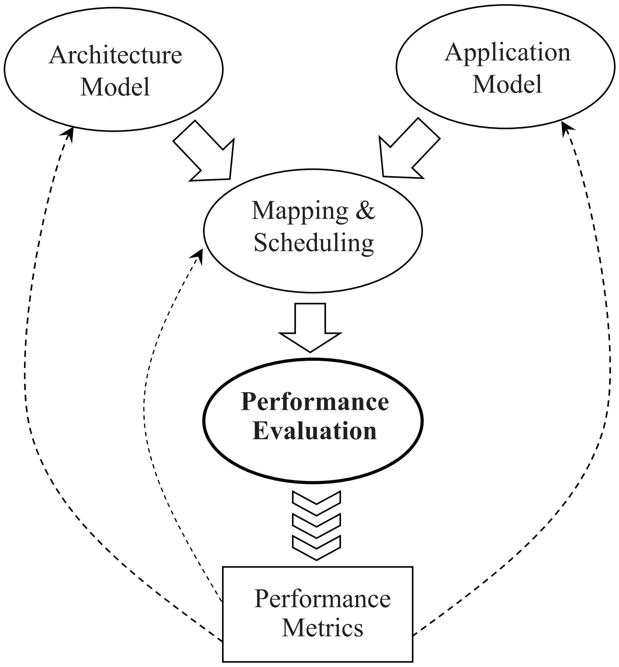


Fig. 1: Typical design space exploration cycle (Y-model)

The figure shows the so-called Y-model, which is largely employed for design space exploration in hardware/software co-design. It represents an iterative design flow with successive refinements. The design starts from modelling the hardware platform (architecture) and the software structure (application) separately. The next design step consist in deciding the assignment of software tasks to processing resources (mapping) and determining the precedence relations among tasks (scheduling). At this point a complete system model is configured, and its performance can be ascertained by an appropriate performance evaluation method. After the analysis, the performance results are used to derive the next exploration step in the design space. In particular, based on the obtained performance metrics, the design choices in terms of architecture, application, as well as mapping and scheduling are revised.

It becomes apparent that the system-level performance evaluation plays a central role in the described design methodology. It guides the designer to efficient solutions in a potentially huge design space, and thus supports him in taking crucial design decisions early in the design flow.

Finally, we would like to note that, in general, not only the mentioned performance metrics, but also other design characteristics such as software code size, chip area, cost, etc. are considered in the exploration of the design space. In the present thesis we will, however, focus solely on the performance metrics listed above.

1.3.2 Requirements

There are several requirements that an ideal method for system-level performance evaluation should fulfil. In the following, we specify a list of requirements for performance evaluation methods adapted from [TW].

- ***Modelling Scope***

  The modelling capabilities of the evaluation technique should be rich, meaning that the method should cover a broad domain of systems. In particular, the evaluation technique should be able to precisely model a large variety of processing and communication components, scheduling policies, and arbitration protocols.

- ***Correctness***

  The results of the performance evaluation should be correct in the sense that the determined performance bounds should be inviolable by the modelled system. In other words, every behaviour that the system may exhibit should be contained in the performance bounds resulting from the analysis. Note that correct performance evaluation methods are often also denoted as *conservative* or *exhaustive* methods.

- ***Accuracy***

  The result of the performance evaluation should be accurate, meaning that it should be as close as possible to the actual worst-case or best-case performance of the system. In other words, pessimistic analysis results should be avoided.

- ***Fast Evaluation***
  The run-times of the evaluation tools should be short. In particular, this is essential if the performance evaluation is used in a design space exploration loop as described in Section 1.3.1.

- ***Scalability***
  A method for performance evaluation should not just be applicable to small example systems, but should scale to large, industrial embedded systems.

- ***Modularity***
  The evaluation method should be modular in the sense that the designer can model and analyze a system by composing several smaller, ideally pre-build system modules. The modularity is a key requirement for an analysis method, as it enables the fast reconfiguration and extension of existing models, as well as the incremental design of systems.

Note that some of the listed requirements might be conflicting. For instance, a more accurate technique for performance evaluation often comes at the price of a slower analysis. Finally, we would like to point out that the relevance of the individual requirements highly depends on the particular application domain and design method.

1.3.3 Approaches

In this section we give a rough overview of approaches to system-level performance evaluation of embedded systems, and describe the major pros and cons of each methodology. For a detailed discussion of particular techniques, we refer the reader to Chapter 2.

Most contemporary methods for performance evaluation fall in one of the three main classes reported in Table 1. The first class is formed by empirical evaluation techniques such as simulation, testing or measurements (on prototypes or real systems). Simulation-based methods are often characterized by rich modelling capabilities, meaning that they can represent a broad domain of systems in large detail. However, as common for empirical methods, they are typically not exhaustive. In particular, every simulation run is of finite length and, hence, simulations can reproduce only a finite set of system behaviours. This implies that, in general, corner cases might be missed by a simulation. In terms of performance evaluation, this means that simulation-based methods cannot be employed to derive hard performance guarantees for a system, as the delivered worst-case (best-case) performance bounds might be incorrect. This is represented in Figure 2, which illustrates the typical distribution

	Empirical Methods (Simulation, Testing, Measurements)		Analytical Methods (Mathematical System Abstractions)			State-based Verification Methods (Model Checking)
+	Large modelling scope	+	Exhaustive		+	Exhaustive
-	Not exhaustive	+	Fast		+	Accurate (exact)
(-)	Slow	-	Limited modelling scope	Abstraction loss	-	Slow (State space explosion)
		-	Limited accuracy			

Tab. 1: Methodologies for performance evaluation

of bounds obtained with different techniques for a generic performance metric.

Another category of methods for performance evaluation are analytical techniques. They are based on mathematical abstractions of system behaviours and provide closed-form expressions to quantify the performance of a system. Analytical approaches are often considerably faster than simulation-based methods, but more importantly, they are provably correct. For this reason, they can be used to safely bound the performance of a system. However, analytical methods for performance evaluation often suffer from limited modelling scope because they rely on a restricted set of component models and are limited to the analysis of specific performance metrics. If the system to be analyzed does not closely fit this set of models, it might still be possible to find a conservative approximation of the system behaviour, but at the price of reduced analysis accuracy. A second issue is that the mathematical abstractions employed by these methods might themselves not be tight. In other words, even if a system matches the modelling capabilities of an analysis method, the performance bounds derived for it might still be overly conservative. These two kinds of accuracy problems are commonly referred to as *abstraction loss* of an analytical performance evaluation method. Figure 2 shows the qualitative impact of the abstraction loss on the performance bounds.

The third family of approaches for performance evaluation are the state-based verification methods. They mostly rely on techniques from the domain of model checking. In particular, they offer some formalism for specifying a model of a system, and employ a model checker to automatically verify whether the model meets a given (performance) property, typically specified as formula in (temporal) logic. Compared to analytical approaches for performance evaluation, these methods have a

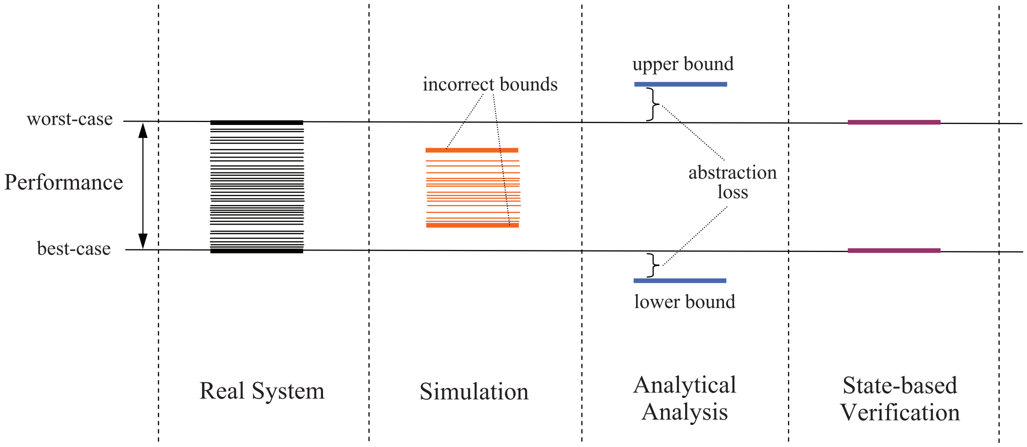


Fig. 2: Typical performance bounds obtained with different evaluation methods (adapted from [Wan06])

much larger modelling scope, as they can typically model arbitrary state-dependent behaviour of system components. Moreover, they can guarantee exact performance results, meaning that the results are not only correct but also perfectly accurate (cf. Figure 2). Unfortunately, state-based verification methods suffer from state-space explosion, a problem that severely inhibits their practical application. More specifically, the state-transition system that is derived from a high-level model grows very quickly with the size of a model. The consequence is that for large-scale models, the verification tools often exhibit prohibitively long verification times and/or large memory requirements.

Finally, we would like to mention that not all existing techniques for performance evaluation fit in one of the above classes. For instance, there are also several stochastic approaches which we do not consider in the scope of this thesis.

1.4 Thesis Overview and Contributions

This thesis focuses on analytical, modular methods for performance evaluation of distributed embedded systems. Its principal aim is to fight the abstraction loss of these methods. In particular, our work improves the application scope and the accuracy of a specific formalism for performance evaluation, the framework for *Modular Performance Analysis* (MPA) [TCN00, WTVL06]. Most of the abstractions and concepts introduced in this thesis are, however, also applicable to other analytical formalisms.

In the following we summarize the contents and the individual contributions of the five main chapters of the thesis.

Chapter 2: Formal Methods for Performance Evaluation

In Chapter 2, we provide a survey and general discussion of existing formal methods for performance evaluation of distributed real-time systems. The survey is followed by an introduction of the framework for Modular Performance Analysis, which forms the basis for the theoretical contributions of this thesis.

The second part of Chapter 2 is devoted to the quantitative assessment of different methods for performance evaluation. In particular, we reproduce and extend the results presented in [Per06], where we investigated the influence of different abstractions on the accuracy of the performance evaluation. The assessment is based on a set of benchmark systems that are used to quantify the accuracy and the runtimes of various evaluation tools. The comparison points out several pitfalls for analytical performance evaluation which stimulated most of the research presented in this thesis.

Chapter 3: Cyclic Dependencies

In Chapter 3 we approach a major obstacle for modular, analytical techniques for performance evaluation, namely the analysis of models with cyclically dependent components. In particular, we extend the modelling scope of MPA to systems with non-functional cyclic dependencies by showing that such systems can be safely analyzed by means of fixpoint iteration. While this is a natural approach successfully used in many other domains, in the context of MPA (and similar methods), it was unclear to what extent the resulting fixpoints are faithful to the performance of the modelled systems. Moreover, it was not clear how to best choose starting points for fixpoint iterations in MPA. We solve these problems by providing the theoretical foundations for fixpoint iterations in MPA. More specifically, Chapter 3 contains the following contributions:

- We develop a general operational semantics underlying the MPA framework.

- On this basis, we prove central properties about the faithfulness of fixpoint computations in MPA.

Chapter 4: Structured Event Streams

In Chapter 4 we introduce models and methods that considerably reduce the abstraction loss of MPA by extending its modelling scope to a highly relevant design pattern: the merging and splitting of event streams in stream-based distributed embedded systems, based on event type information. This pattern applies, for instance, if data from different streams is first combined, transmitted over a shared communication channel, and then separated again. Our model is based on a novel characterization of structured event streams which seamlessly integrates in the existing framework for Modular Performance Analysis. In particular, the contributions of Chapter 4 can be summarized as follows:

- We propose a new approach for analyzing the processing and communication of merged event streams in distributed embedded systems. The approach is based on Event Count Curves, a model for representing structures in event streams.
- We show how the FIFO component introduced in Chapter 2 can be used to handle structured event streams.
- We evaluate the two proposed models and compare their performance with existing techniques from related work. We also apply the proposed techniques to a realistic application scenario.

Chapter 5: Hybrid Performance Evaluation

In Chapter 5 we introduce a novel hybrid methodology for the performance evaluation of distributed real-time systems. The approach combines analytical and state-based performance analysis. In the resulting hybrid and modular framework, system components can be modelled by either MPA or Timed Automata, a state-based formalism for the verification of real-time systems. In this way, we can benefit from the advantages of both domains: On one hand, we obtain considerably better modelling capabilities and analysis accuracy compared to a pure MPA representation of a system. On the other hand, we can avoid state space explosion by constraining the verification scope to the level of single system components. The interfaces among components in the hybrid analysis methodology rely on conversions of arrival curves (the event stream model adopted by MPA) to networks of co-operating Timed Automata and vice versa. To summarise, the following contributions are contained in Chapter 5:

- We describe a pattern allowing us to convert arrival curves (or other common event stream models) to networks of co-operating Timed Automata, and vice versa.

- We prove the correctness and tightness of the proposed transformations, i.e., the Timed Automata generate all event traces and solely the event traces complying with the arrival curves.
- We evaluate the accuracy and scalability of the proposed methodology.

Chapter 6: MPA for Energy-Efficient System Design

While the Chapters 2-5 focus mainly on the analysis of timing characteristics of embedded systems, in Chapter 6 we approach the analysis of another very important performance aspect: the worst-case energy consumption of embedded systems. More specifically, we look at the combined observance of real-time and energy constraints. We show how Dynamic Voltage Scaling (DVS), a common technique for the reduction of the energy consumption of a processor, can be seamlessly incorporated in the framework of MPA, extending its application scope to the design of energy-efficient real-time systems. In particular, we consider two different design problems: (1) The energy-efficient *static* assignment of execution speeds and priorities to a set of event streams. (2) The energy-efficient *dynamic* adjustment of the execution speed to process a single event stream. The first method is based on the original MPA framework for performance evaluation, whereas the second one builds on top of the hybrid evaluation framework introduced in Chapter 5. Specifically, the following contributions can be identified for Chapter 6:

- We devise algorithms to derive energy-efficient static priority and speed assignments to multiple real-time tasks with arbitrary release patterns.
- We present an adaptive scheme that dynamically adjusts the execution speed to process an arbitrary event stream. We show that the scheme guarantees timing and speed constraints, and at the same time ensures energy-efficient processing.
- We demonstrate the effectiveness of the presented methods by means of experimental test cases.

2

Formal Methods for Performance Evaluation

The performance evaluation is a central step in the design process of an embedded system. Testing and measurements on prototypes or real systems are largely applied to validate the performance of system implementations. However, a major drawback of these methods is that they cannot be applied at early design stages, that is, when implementation details of the system are not yet defined. Model-based techniques are an alternative way to evaluate crucial (performance) characteristics of a system early in the design flow. The principle of model-based engineering is to construct simplified models of a system in order to reason about its properties. The major challenge lies in constructing models which are abstract enough to cope with missing implementation details, but nevertheless allow us to draw correct conclusions about the system. There is a large body of work on model-based design and verification of embedded systems. In the present chapter we first provide an overview of existing formalisms, with a focus on abstraction methods for system-level performance evaluation (Section 2.1). In Section 2.2 we detail the theoretical foundations of the Modular Performance Analysis (MPA), the formalism used as basis for the contributions of this thesis. Finally, in Section 2.3 we reproduce and extend our work of [Per06], that quantitatively compares several approaches to performance evaluation. The comparison identifies various pitfalls for analytical performance evaluation, and forms the basis for the extensions and improvements presented in the following chapters.

2.1 Overview of related work

There are several different basic principles for model-based performance evaluation of embedded systems. An important distinction is drawn between empirical and worst-case methods for performance evaluation. The former class includes most simulation-based approaches, whereas the latter contains exhaustive methods such as analytical or state-based verification techniques. Besides these two classes there are also stochastic methods for performance evaluation which we do not, however, consider in this context.

The use of simulation to estimate the performance of system designs is the current state-of-the-art in many application domains. There exist various simulation-based methods for different levels of abstraction. Commercial tool suites for electronic system design offered by companies such as Cadence, Synopsis, Mentor Graphics, or Magma offer a broad range of simulation instruments from high-level discrete event simulators to cycle-accurate software/hardware co-simulators. Apart from commercial design tools, there are also open-source simulation environments. Examples are processor architecture simulators such as SimpleScalar [ALE02, Sim] and PTLsim [You07, PTL], network simulators such as ns-2 [ns2], full system simulators like OVPsim [Ovp], and system-level discrete event simulators such as SystemC [Sys, GLMS02]. While most of these simulators have a fairly broad application scope, there are also tools dedicated to particular system types, e.g. the TrueTime simulator for networked control systems [CHL+03, TT].

The main advantage of simulation-based methods for performance evaluation is their large and extendable modelling scope. In other words, they permit to represent a broad domain of systems with an almost arbitrary level of detail. Unfortunately, detailed system simulations are often very time consuming, especially if high timing fidelity is required (e.g. cycle-accurate simulators). The crucial drawback of simulation-based approaches is that they are typically not exhaustive, that is, they cannot guarantee full coverage of the system behaviour. Especially for large and complex system architectures, it is often unfeasible to reveal performance corner cases by means of simulation. The consequence is that simulators cannot be employed to provide guarantees on the best-case/worst-case performance of embedded systems, as required by several application domains (e.g. hard real-time systems).

The need for reliable and provably correct performance bounds for complex embedded systems has driven research for many years. The result are several analytic and state-based formal methods for worst-case performance evaluation, which are discussed in the remainder of this section.

Various analytical methods for timing/performance evaluation have been introduced for different abstraction levels in the design flow. At the process level, designers of hard real-time systems are typically interested in determining the worst-case execution time (WCET) of programs. The complexity of this task depends largely on the architecture of the underlying processor, as components such as caches, pipelines, or branch predictors complicate the analysis. Analytical methods for WCET analysis have been proposed in [LM95], [Wil] amongst others. For an extensive overview of methods and tools we refer the reader to [WEE+08].

At the system level, designers have to guarantee the performance of multiple real-time tasks that are executed on a single processor or on a distributed system. The analysis of the worst-case completion times of concurrent tasks on a single processor is commonly denoted as *scheduling analysis*. This field has been extensively studied over the past 35 years and there is a large body of results for different task scheduling policies such as Rate Monotonic Scheduling (RM) [LL73], Fixed Priority Scheduling (FP) [JP86, TBW94], Earliest Deadline First Scheduling (EDF) [LL73], or Round Robin Scheduling (RR) [RKZ95, RLH+07]. There also exist many extensions for the various scheduling algorithms, e.g. for the consideration of offsets [PG98], mutually exclusive shared resources [BA06], task re-executions [LB03], and limited priority levels [BYJ03]. Detailed information about the various algorithms can be found in [But97], as well as in other books on the topic. In recent years, the attention of the real-time scheduling community has shifted mostly to multiprocessor platforms, e.g. [GFB03, BBMS10].

The analytic performance evaluation of *distributed* embedded systems has also been investigated. The major intricacies with respect to the single processor case are concurrent task executions, heterogeneous resources as well as communication delays on shared communication devices. Several methods have been proposed so far for the worst-case analysis of distributed real-time systems. The methods are based on essentially different abstractions. The first idea was to extend results of the classical scheduling theory to the distributed case. The resulting combined analysis of processor and bus scheduling is often referred to as *holistic scheduling analysis*. Rather than a single evaluation method, holistic scheduling denotes a collection of techniques for the analysis of distributed systems, each of which is tailored for a specific combination of processor scheduling and communication arbitration. The first work in this direction is by Tindell and Clark. In [TC94] they combine FP preemptive scheduling on the processors of a distributed system with TDMA scheduling on the interconnecting bus. In [PEP02] Pop, Eles and Peng analyze mixed event-triggered and time-triggered task sets that communicate over protocols with both static and dynamic phases such as FlexRay. Holistic scheduling under the presence of data or control dependencies has been studied in [YW95] and [PEP00], respectively. Other holistic analysis techniques can be found in the literature, e.g. [PG03]. The major drawback of holistic analysis methods is their poor flexibility. While they produce accurate results for a particular system architecture, they are typically not applicable to other system configurations. Hence, their modelling scope is strongly limited. This implies that for every new system configuration a new holistic analysis method would have to be developed, which is obviously a very difficult task. Another important issue for holistic techniques is scalability. This is because the complexity of the monolithic models obviously increases with the size of the represented system. The heterogeneous set of available techniques also makes it difficult to apply holistic scheduling in practice. This latter problem was, however, alleviated by González Harbour et al. with the release of the MAST tool [GGPD01], which enhances, implements and aggregates several holistic analysis algorithms.

Compositional or modular analysis methods such as SymTA/S [HHJ$^+$05] and Modular Performance Analysis (MPA) [CKT03] form a more flexible and scalable alternative to analytically bound the worst-case performance of distributed systems. These methods make use of a modular system abstraction in which every hardware/software unit of the system is modelled by an individual abstract component. These abstract components represent single computation or communication tasks of the system. Typically, it is assumed that each task is triggered by the arrival of an input event, and that it produces an output event after its execution is completed. The dataflow among components is then abstracted by means of event streams (timed sequences of events). The basic concept of compositional performance analysis methods is to evaluate the system component-wise by forwarding the output streams of an individual component to other dependent components. System performance metrics such as end-to-end delays of events are then derived by aggregating the results of individual component analyses.

The SymTA/S analysis approach was introduced by Richter, Jersak and Ernst [RJE03, Ric05, Jer05]. The general idea of the methodology is to reuse algorithms from classical scheduling analysis (or extensions thereof) at the component level, and to propagate local analysis results to other components through appropriate interfaces. These steps are repeated until all components in the model are analyzed. In the case of cyclic dependencies among components, fixpoint iteration is applied. Details on the generation of starting points and on the convergence of the SymTA/S analysis in the presence of cyclic dependencies can be found in [Ric05] and [SDI$^+$08], respectively. For the local performance analysis of components, e.g. for determining the worst-case response time to input events, SymTA/S uses formal analysis methods based on the busy window technique proposed by Lehoczky [Leh90]. SymTA/S offers local analysis techniques for FP scheduling (preemptive and non-preemptive), Round Robin, TDMA, EDF, CAN, and several arbitration schemes used in the automotive domain. The key element of the SymTA/S analysis methodology is the use of event models to represent the dataflow among components. An event model is an abstract representation for the timing of event arrivals in a stream. Since most algorithms of classical scheduling analysis assume simple patterns for the arrival of input events (e.g. periodic events, periodic events with jitter), in the original SymTA/S approach [HHJ+05], the interface among components is limited to a small set of simple event models. These are commonly denoted as Standard Event Models (SEM) or Periodic with Jitter (PJD) models. To guarantee that the output event model of a component fits the input event model expected by the following component, the SymTA/S method applies appropriate Event Model Interfaces (EMIF) or Event Adaption Functions (EAF). While the restriction to SEMs enables the reuse of previous results of real-time research, and simplifies the calculation of output event models, it has the drawback of limited analysis accuracy. This is because, on the one hand, complex timing behaviours of general event streams cannot be captured by SEMs and, on the other hand, event model conversions may be lossy. Recently, the restriction of SymTA/S to SEMs was relaxed by Schliecker et al., who extended the methodology to arbitrarily shaped event models [SRIE08]. In particular, the authors of [SRIE08] introduce the multiple event busy time model which they use to characterize the outputs of arbitrarily triggered components under FP scheduling. Many other extensions and ameliorations have been presented for the SymTA/S analysis method. Examples are the consideration of correlations among events (context-based analysis) [JHE04, HE05], pipelined delay analysis [SE09], analysis of MPSoC architectures with shared memory [NSE09], analysis of hierarchical communication [RE08], as well as SymTA/S-based sensitivity analysis [Rac09] and robustness optimization [Ham08]. The rich set of available extensions as well as the development of a commercial tool [Sym] make SymTA/S a very powerful framework for worst-case system-level performance evaluation. Nonetheless, the method has some inherent limitations. First, it is based on existing algorithms of scheduling analysis which means that for any new scheduling policy a new, dedicated analysis needs to be conceived. Moreover, the method is not modular in terms of processing or communication resources. For instance, it does not support the modelling of hierarchical scheduling policies.

A different approach to compositional performance analysis that does not rely on classical scheduling theory is the Modular Performance Analysis (MPA) which was introduced by Thiele et al. in [TCN00]. The method uses the Real-Time Calculus (RTC), a formalism that has its roots in Network Calculus [LT01], to analyze the flow of event and resource streams through a network of computation and communication components. The MPA framework is discussed in detail in Section 2.2.

Apart from analytical methods for performance evaluation, one can also apply general state-based verification techniques to characterize the worst-case performance of a system. The state-based formal verification of systems has been investigated extensively over the past 30 years. The task of automatically verifying whether a state-based model of a system meets a given specification, typically given as formula in temporal logic, is commonly denoted as model checking. Since the pioneering work of Clarke, Emerson and Sifakis [CES86, QS82], various techniques for model checking have been proposed and implemented in verification tools, e.g. [Spi, NuS]. For details on model checking and an overview of existing techniques we refer the reader to [BK08, CGP99]. For the domain of real-time systems, the automatic verification of hybrid models is particularly interesting. Hybrid models are state-based representations of systems that combine discrete system properties with continuous quantities such as time. A prominent example are Timed Automata (TA) [AD94]; similar formalisms can be found in [Hen96, GMM90]. Two examples of corresponding model checking tools are Uppaal [BLL$^+$] and Kronos [BDM$^+$]. While none of the mentioned state-based verification techniques is particularly conceived for performance evaluation, there has been work on applying them for this purpose. For instance, TA modelling techniques have been applied for monoprocessor scheduling analysis, see [FPY02], and the corresponding tool Times [AFM$^+$]. In [NM09, Ch.4] Larsen et al. provide a framework based on Uppaal for the analysis of multiprocessor scheduling scenarios which also considers timing uncertainties and task dependencies. An alternative framework for scheduling analysis of MPSoC systems has been proposed by Brekling, Madsen et al., see [BHM08] and [NM09, Ch.5]. As last example, we mention the approach for worst-case performance evaluation of distributed systems presented by Hendriks and Verhoef [HV06] which relies on customized TA representations of Standard Event Models.

State-based performance verification methods have two advantages with respect to analytical approaches. They typically offer a larger modelling scope, as arbitrary finite-state models can be used to represent the system behaviour. Moreover, they can derive *exact* performance bounds for the modelled system, as there is no inherent abstraction loss for these methods. In other words, state-based approaches for performance verification are typically more accurate than analytical ones. However, the detailed modelling capabilities do not come for free: the verification of state-based models is seriously affected by state-space explosion. This fact often forecloses the application of state-based methods to large, realistic systems.

There are also hybrid approaches for system-level performance evaluation which integrate different paradigms. For instance, in [LRD04] Lahiri et al. present a method that combines simulation for parameter estimation with analytical system evaluation. The main advantage of this method are faster system evaluations with respect to a purely simulative approach. Similar hybrid methods can be found in [KPBT06, BPN$^+$04, HKH$^+$09] and in [NM09, Ch.4]. In Chapter 5 of this thesis, we will discuss a hybrid technique for worst-case performance evaluation that combines MPA and TA, that is, an analytical and a state-based technique.

Finally, we would like to mention languages or frameworks for the compositional design of heterogeneous real-time systems such as BIP [Bip], Ptolemy II [Pto], Metropolis [Met], POOSL [Poo], or Moses [Mos]. They not only offer useful features such as deadlock analysis, correctness by construction, or automatic code generation but also support the simulation and, in some cases, the formal verification of the modelled systems. Hence, they can also be employed for system-level performance evaluation.

2.2 Modular Performance Analysis

In this section, we introduce the framework for Modular Performance Analysis (MPA) which forms the basis for the contributions of this thesis. MPA is an analytical method for worst-case performance evaluation of distributed systems. In contrast to SymTA/S, its origins do not come from classical real-time scheduling, but from network performance analysis. The MPA formalism provides an elegant way of capturing the workload imposed on a system by concurring applications, as well as the service offered by the system architecture. A key feature of MPA is modularity; MPA represents each hardware/software unit of a system by an individual abstract component. This considerably eases the analysis of large and heterogeneous distributed systems. In MPA, abstract components are combined to form a performance model (or MPA model) of a system. An MPA model is used to derive performance metrics like worst-case end-to-end delays or buffer fill levels. What distinguishes MPA from other analysis methods is that it explicitly characterizes the service, i.e. the processing or communication resources, which are available to individual components. This not only permits systems designers to model complex resource availability patterns, but also strengthens the compositionality of the approach in terms of resource sharing schemes.

In the following, we first illustrate the general concepts of the modular system abstraction employed in MPA (Section 2.2.1). Then, we briefly describe a simple formalism for characterizing the performance of a system in the time domain (Section 2.2.2). As we will explain, this formalism in the concrete time domain is of limited practical use for the worst-case performance evaluation of systems. However, it serves as basis for the introduction of the Real-Time Calculus (Section 2.2.3), the interval-based formalism commonly used within MPA for worst-case performance evaluation.

2.2.1 A general performance model

We will illustrate the general modelling approach of MPA by means of a simple example. Consider the system design shown in Figure 3. The system consists of a distributed architecture that processes two concurrent data streams. The data streams consist of data tokens that are generated by the data sources *I1* and *I2*. Both data streams are first processed by dedicated tasks on *CPU1*, then transmitted over a bus, and finally processed further by the tasks of *CPU2*. The processed data tokens are collected by the sinks *O1* and *O2*. In the considered system, each processing or communication unit, i.e. each task on a CPU and each channel on the bus, has a dedicated input buffer which temporarily stores data tokens if the corresponding unit is already busy. We assume that a preemptive fixed priority scheduler is adopted on *CPU1* and *CPU2* to execute the respective tasks. In particular, we let task *T1* have priority over task *T2*, and task *T3* have priority over task *T4*. For the communication bus, we suppose that a TDMA arbitration scheme is employed to schedule channels *C1* and *C2*. We assume that the data generation patterns of the data sources are known, as well as the worst-case execution (transmission) times of tasks (channels). The goal of the analysis is to safely bound performance characteristics of the system such as the maximum end-to-end delays of data tokens, or the maximum buffer fill levels.

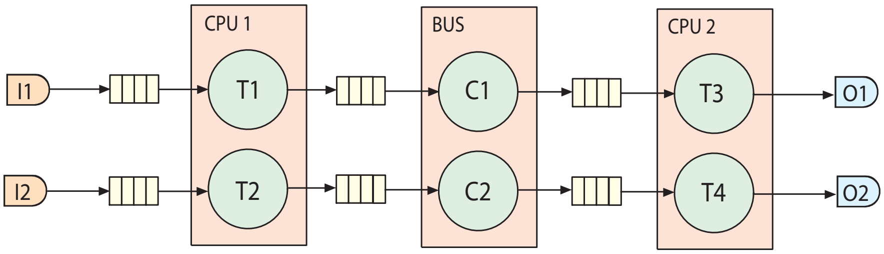


Fig. 3: Architecture of example system

Figure 4 shows the MPA model for the described system. MPA adopts component networks as a basic model of computation. Each component represents an individual processing or communication unit of the modelled system. The components communicate via event passing. In particular, the communication is based on infinite FIFO (first-in first-out) event buffers among the components.[1] The components are triggered by input events which represent incoming data tokens that need to be processed. Outgoing data tokens are modelled by means of output events. These events form *event streams*, that is, potentially infinite sequences of timed events. The abstraction of event streams is used in MPA (and other modular formalisms such as SymTA/S) to represent the dataflow in a distributed system. In Figure 4, the event streams are represented by horizontal arrows.

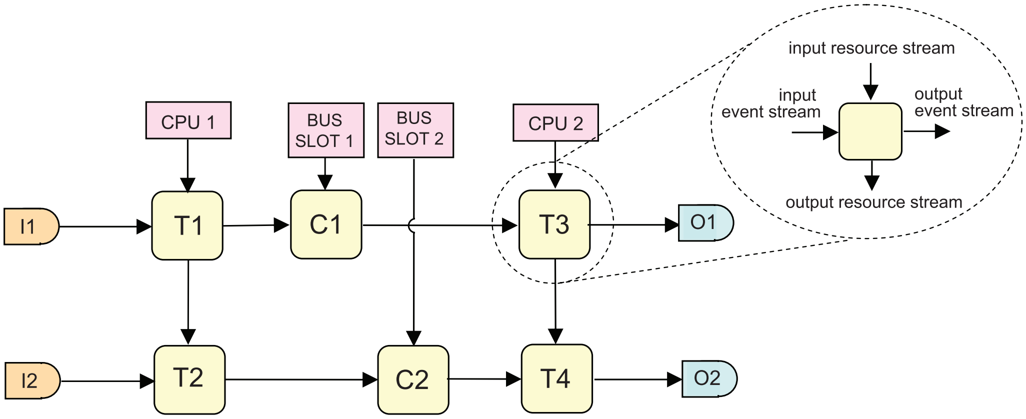


Fig. 4: General MPA model of example system

The MPA formalism also explicitly quantifies the service that is available to individual components in terms of processing or communication resources. This is done by means of *resource streams*. In particular, an input resource stream is used to represent the availability of a resource to a component over time. Similarly, an output resource stream models the resources that are left over by a component. In Figure 4, the resource streams are represented by vertical arrows. Note that scheduling policies for shared resources are typically modelled by the way the resource streams are propagated. For instance, to represent the fixed priority scheduling policy adopted on *CPU1* for the tasks *T1* and *T2*, we connect the resource output of *T1* with the resource input of *T2*. In other words, *T2* gets only the processing service left over by *T1*. An important difference between event and resource streams is that the service cannot be buffered. In other words, resources that are not immediately used by a component are wasted.

---

[1] For the sake of simplicity, these event buffers are not shown in Figure 4 and in all the other MPA models represented in this thesis.

To summarise, in the abstract view of MPA, distributed systems are networks of components that manipulate event and resource streams. More precisely, the individual components model the interplay over time between demanded and available system resources. Hence, the performance model is well suited to derive local and system-wide performance metrics such as latencies or buffer demands.

Next to the general view of performance models in MPA, we obviously also need a (mathematical) formalism that specifies the following points:

- How event and resource streams are represented
- What the transfer functions of different components are
- How performance metrics (e.g. end-to-end delays) are computed

These points will be addressed in Sections 2.2.2 and 2.2.3, where we present two different but closely related alternatives for performance analysis within MPA.

2.2.2 Performance characterization in the time domain

The most natural way to derive the performance of a component network in MPA is to specify the involved streams as well as the transfer functions of the various components in the time domain. In the following, we introduce a simple formalism that allows us to do so.

2.2.2.1 Arrival and service functions

An event trace is, in essence, a timed sequence of event arrivals. More precisely,

- a *timed event* is a pair (e, t) where e is some event and $t \in \mathbb{R}$ an associated time stamp;
- an *event trace* (or *concrete event stream*) is an (infinite) sequence $(e, t_1); (e, t_2); \ldots$ of timed events which are ordered by non-decreasing time stamps, i.e., $t_i \leq t_{i+1}$ for $i \geq 1$.

In the following, we use the notion of arrival function to characterize event traces.

Def. 1: **(Arrival Function)** *An arrival function* $r : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}^{\geq 0}$ *unambiguously represents an event trace, where* $r(s, t)$ *for* $s < t$ *denotes the number of events that arrive in the time interval* $[s, t)$*, with* $r(s, s) := 0$.

In simpler words, the arrival function r 'counts' the number of event arrivals in a given time interval.[2] Note that the definition of arrival function is not necessarily bound to the above concept of event trace: For a given time interval, an arrival function can also quantify the demanded processing cycles, the bytes to transmit, or any other discrete or continuous workload unit.

For interpretation, it is often useful to consider the following property which holds for any arrival function as defined above:

$$r(s,t) = r(a,t) - r(a,s) \qquad \forall a \leq s \leq t \tag{2.1}$$

For instance, if only the positive time axis is considered for the arrival of events, one may choose $a = 0$ in Equation (2.1).

As mentioned above, components typically need resources in order to perform the requested operation. We describe the availability of resources by means of service functions.

Def. 2: **(Service Function)** *A service function* $c : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}^{\geq 0}$ *unambiguously represents a concrete resource stream, where* $c(s,t)$ *for* $s < t$ *denotes the number of resource units that are available in the time interval* $[s,t)$ *with* $c(s,s) := 0$.

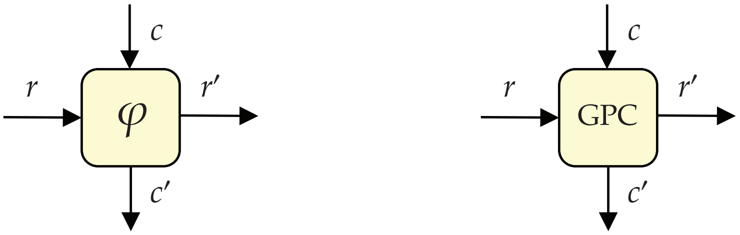
Again, we allow a liberal definition of resource units here. The amount of available resources can be specified in resource token, processing cycles, bytes, or any other discrete or continuous quantity.

2.2.2.2 Component models

The basic building blocks of performance models are *performance components*. Performance components can be basically seen as deterministic transducers of concrete event and resource streams. More precisely, a general performance component φ receives an input event trace r, and an input resource stream c, and it produces an output event trace r', and an output resource stream c'. This view is illustrated in Figure 5(a). In the figure, we use φ to denote the behaviour of the component. φ is a transfer function from input to output, i.e., $(r', c') = \varphi(r, c)$. The function φ reflects the particular processing semantics of the component. For now, we consider only performance components with a single event input port and a single event output port. However, MPA also supports performance components with multiple event input/output ports (cf. Section 2.2.3).

---

[2] For easier readability, in this thesis we use the letter r to denote both, an event trace and the corresponding arrival function; it is clear from the context to which of the two we refer.



(a) A general performance component (b) Greedy Processing Component

Fig. 5: Performance components

Various components with different behaviours comply with the above general description. One particular performance component which represents the behaviour of many hardware or software units is the *Greedy Processing Component (GPC)*. It models a task that is triggered by the events of the incoming event stream which queue up in a FIFO buffer. The task processes the events in a greedy fashion while being restricted by the availability of resources. In other words, the task processes an event as soon as it receives resources to do so. The graphical representation of a GPC is shown Figure 5(b). Note that often we also denote GPCs with the name of the task they model (cf. Figure 4).

Thm. 1: **(Transfer function GPC)** *A GPC component has the input/output relations*

$$r'(s,t) = \inf_{s \le \lambda \le t} \{ r(s,\lambda) + c(\lambda,t) + b(s),\, c(s,t) \} \tag{2.2}$$

$$c'(s,t) = c(s,t) - r'(s,t) \tag{2.3}$$

where $b(s)$ *denotes the initial fill level of the event input buffer.*

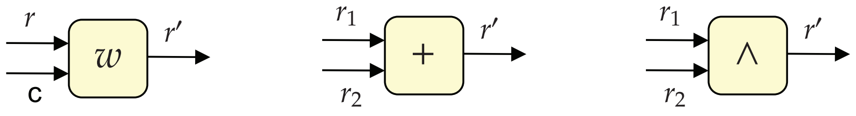
Proof. In any time interval, the output of the component is restricted by the available resources. Hence, we have $r'(s,t) \le c(s,t)$ and also $r'(s,t) \le r'(s,\lambda) + c(\lambda,t)$. But at the same time, in any time interval the component cannot output more events than those available at the input. Thus, we have $r'(s,\lambda) \le r(s,\lambda) + b(s)$. If we combine these constraints, we obtain $r'(s,t) \le \min\{r(s,\lambda) + c(\lambda,t) + b(s), c(s,t)\}$. Let us now assume that there is some last time $\lambda^* < t$ when the input buffer was empty. This implies that all events that arrived up to λ^* are processed by that time, i.e. $r'(s,\lambda^*) = r(s,\lambda^*) + b(s)$. In the interval (λ^*, t) the input buffer is never empty, which means that all available resources are being used to produce output events, i.e. $r'(s,t) = r(s,\lambda^*) + b(s) + c(\lambda^*,t)$. In the case that the buffer is never empty, we simply have $r'(s,t) = c(s,t)$. As a result, we obtain the transfer function (2.2). On the other hand, relation (2.3) is obvious: the component cannot buffer resources over time. Hence, in any time interval, the remaining resources are simply given by the available resources minus the consumed resources.

□

An important prerequisite for the above relations is that the workload r and the service c are specified in the same unit. Let us, for instance, assume that r counts the number of arriving input events, but c quantifies the available service in processing cycles. In this case, the mismatch can be resolved by determining the resource demand of each input event in terms of processing cycles, i.e. by converting r to a resource-based representation. In the simplest case, each input event imposes the same workload on the component. Then, the conversion boils down to a simple scaling of the arrival function r (see the Scaler component below).

Besides performance components which specify the interplay of event and resource streams, a performance model can also contain simpler components that do not interact with resources. The following list briefly describes a few such *elementary components*. The corresponding graphical representations are shown in Figure 6.

- A *scaler* component is used for converting workload representations (e.g. from events to processor cycles). Its transfer function is given by
$$r'(s,t) = w \cdot r(s,t), \tag{2.4}$$
where w is a positive scaling factor.
- An *OR* component produces an output event for every input event. The transfer function is
$$r'(s,t) = r_1(s,t) + r_2(s,t). \tag{2.5}$$
- An *AND* component produces an output event only if there is an input event on *all* inputs. For two input event streams r_1, r_2 the transfer function of the AND component is
$$r'(s,t) = \min\{b_1(s) + r_1(s,t), b_2(s) + r_2(s,t)\}, \tag{2.6}$$
where $b_1(s)$ and $b_2(s)$ denote the fill levels of the two input buffers at time s.



(a) Scaler (b) OR (c) AND

Fig. 6: Elementary components

We assume that all elementary operations are immediate. In other words, elementary components do not introduce any delay.

2.2.2.3 Performance Analysis

What remains to be specified is how performance characteristics of a system are derived from the performance model. For a GPC component, we can use the following equation to quantify the fill level b of the input buffer at time t, provided that we know the fill level b at initial time s:

$$b(t) = b(s) + r(s,t) - r'(s,t) \tag{2.7}$$

Similarly, we can quantify the delay that an event experiences at a GPC. In particular, the delay d experienced by the last event arriving in the interval $[s,t)$ is given by:

$$d = \inf\{\, \tau \geq 0 : \; b(s) + r(s,t) \leq r'(s, t+\tau) \,\} \tag{2.8}$$

Once the performance of each individual component of a model has been computed, we can derive system-wide performance metrics by aggregating the individual results. For instance, the end-to-end delay experienced by a particular event is simply given by the sum of the delays experienced at the individual performance components that the event traverses.

2.2.2.4 Limitation

Let us now focus on a major limitation of the formalism as presented so far. The fundamental problem of the representation in the time domain is that it works for *concrete* instances of event and resource streams only. In other words, the formalism supposes that the system designer has full, a priori knowledge of the input streams, and that the timing behaviour of all streams in the system is totally deterministic. Unfortunately, this is not the case in real systems where event streams and resource availabilities exhibit large variability in their timing behaviour. For instance, tasks in embedded systems are very often triggered by the physical environment which can, in general, not be predicted accurately. The situation is further aggravated by complex processing or communication components with variable execution times. And finally, the components of distributed embedded systems are often only loosely coupled, which leads to complex interference patterns on shared resources. In a nutshell, real systems present a substantial degree of timing non-determinism (at least from a system-level perspective). For such systems, a worst-case performance evaluation must guarantee to cover *all* possible system behaviours. The discussed formalism can, however, only evaluate *particular* system executions. In fact, it is equivalent to a system simulation, and hence not adequate for deriving worst-case performance bounds for systems with non-deterministic streams.

2.2.3 Real-Time Calculus

In this section, we discuss the Real-Time Calculus (RTC), a formalism conceived for worst-case performance evaluation within MPA. It was introduced by Thiele et al. in [TCN00, CKT03], and has its roots in Network Calculus [LT01, Cha00], a deterministic queuing theory for worst-case characterization of communication networks. RTC does not operate in the time domain, but in the interval domain. More specifically, it uses abstract event and resource streams, an interval-based representation that efficiently captures the variability of arrival and service patterns. To represent the processing and communication units of a system, it employs abstract performance components which operate on abstract streams. By doing so, RTC provides a means to capture *all* possible behaviours of the modelled system. Hence, embedded in the MPA framework, RTC is a suitable formalism for worst-case performance evaluation of distributed systems. In the remainder of this section, we introduce the basic theoretical notions of RTC.

2.2.3.1 Event stream model

RTC abstracts from particular event traces. It uses *abstract event streams* to specify all event traces that can appear in a stream. An abstract event stream simply represents a set of event traces. In RTC, abstract event streams are specified by a tuple of arrival curves $\alpha(\Delta) := [\alpha^u(\Delta), \alpha^l(\Delta)]$ where $\alpha^u(\Delta)$ denotes the upper arrival curve and $\alpha^l(\Delta)$ the lower arrival curve of the stream.

Def. 3: **(Arrival curves)** *Let $r(s,t)$ be the arrival function of any event trace belonging to an abstract event stream. Then, r, α^u, α^l are related to each other by the inequality*

$$\alpha^l(t-s) \leq r(s,t) \leq \alpha^u(t-s) \quad \forall s \leq t \tag{2.9}$$

with $\alpha^l(0) = \alpha^u(0) = 0$. If the above inequality holds for an event trace r, we say that r conforms to *α, denoted as $r \models \alpha$.*

Informally, an upper arrival curve $\alpha^u(\Delta)$ specifies the maximum number of events that can appear in the stream in *any* time interval of length Δ.[3] Similarly, a lower arrival curve $\alpha^l(\Delta)$ specifies the minimum number of events for any time interval of length Δ. Both upper and lower arrival curves are monotonically increasing functions. An example of a tuple of arrival curves that models an abstract event stream is shown in Figure 7.

---

[3] Note that, depending on the definition of the arrival function r, the number of events may refer to a discrete or continuous quantity of workload units.

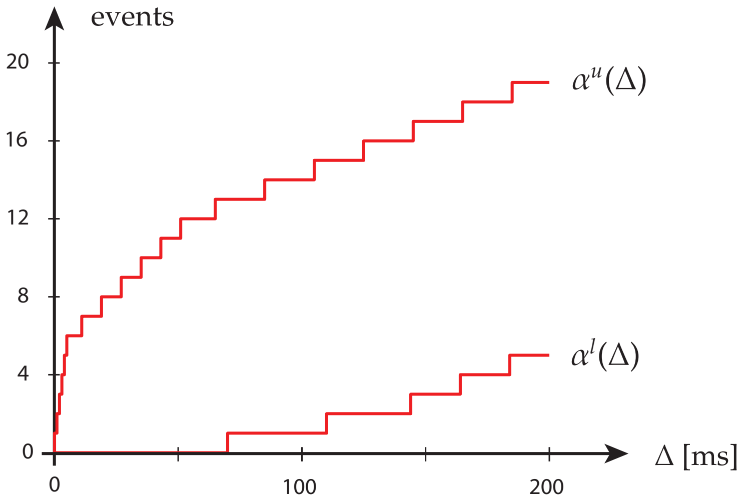


Fig. 7: Tuple of arrival curves (example)

Def. 4: **(Set of conforming event traces)** *Let* α *be a tuple of arrival curves as defined above. The set of all event traces that conform to* α *is defined by*

$$R^{\alpha} := \{r \in R : r \models \alpha\}, \tag{2.10}$$

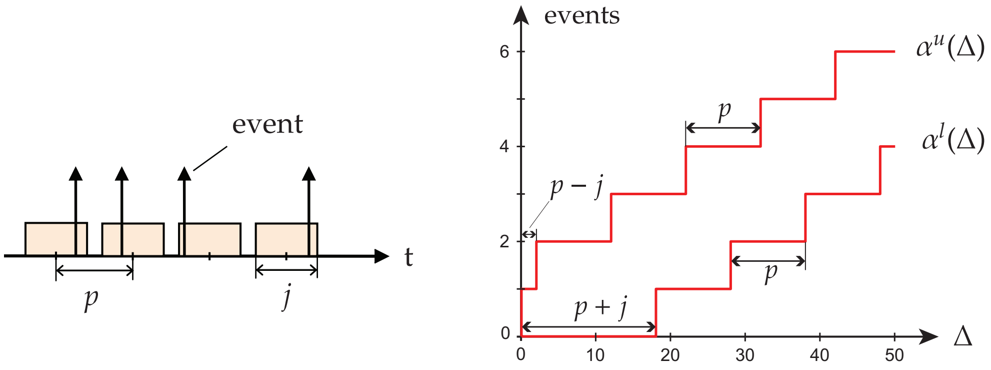
where R *denotes the set of all event traces.*

The conformance of an event trace r to an upper (lower) arrival curve α^u (α^l), as well as the sets R^{α^u}, R^{α^l} are defined accordingly. For a particular event trace specified by an arrival function $r(s,t)$, the tightest arrival curves α^u_r, α^l_r that model the trace are given by:

$$\alpha^u_r(\Delta) = \sup_{s\in\mathbb{R}} r(s, s+\Delta) \qquad \forall\Delta \geq 0 \tag{2.11}$$

$$\alpha^l_r(\Delta) = \inf_{s\in\mathbb{R}} r(s, s+\Delta) \qquad \forall\Delta \geq 0 \tag{2.12}$$

The abstraction of arrival curves is very general, as *any* abstract event stream can be represented by an appropriate tuple $[\alpha^u(\Delta), \alpha^l(\Delta)]$. Arrival curves generalize classical event stream models such as sporadic events, periodic events, periodic events with jitter etc. In particular, arrival curves are more expressive than the *PJD model*, an event stream model commonly used in the literature (also denoted as *Standard Event Model* in the context of SymTA/S). A PJD model is specified by three parameters: p, j and d. It represents an event stream in which events arrive periodically with period p, but may have a jitter of up to j time units around the ideal periodic arrival time. The parameter d specifies the minimum time between consecutive event arrivals. Figure 8(a) shows an example of an event trace conforming to a PJD model with $j < p$ and $d = 0$. The shaded boxes indicate the admissible arrival times of events in the stream (one event for each box). The arrival curves that bound *all* the event traces of the corresponding abstract event stream are represented in Figure 8(b). Note that if $j \geq p$, the jitter boxes overlap. In this case, the stream can contain bursts of event arrivals, i.e., multiple event arrivals at short distance (or even at the same time, depending on the value of d).



(a) Example event trace

(b) Arrival curves of event model

Fig. 8: PJD event model with period p, jitter $j < p$, and minimum event distance $d = 0$.

The derivation of the arrival curves corresponding to a PJD event model is straightforward:

$$\alpha^u(\Delta) = \min\left\{\left\lceil\frac{\Delta + j}{p}\right\rceil, \left\lceil\frac{\Delta}{d}\right\rceil\right\} \quad \forall\Delta \geq 0 \tag{2.13}$$

$$\alpha^l(\Delta) = \max\left\{0, \left\lfloor\frac{\Delta - j}{p}\right\rfloor\right\} \quad \forall\Delta \geq 0 \tag{2.14}$$

On the other hand, a general tuple of arrival curves can typically not be expressed precisely as PJD model. While it is always possible to find a PJD model which conservatively approximates the arrival curves (see [KHET07]), this often leads to less accurate performance results.

We would like to highlight that arrival curves are not necessarily bound to the concept of event traces. Depending on the interpretation of the corresponding arrival function, arrival curves can bound any convenient workload unit in a time interval, e.g., number of demanded processing cycles, number of bytes to transmit, and even continuous arrivals of infinitesimally small workload units. When applying the RTC formalism, it is often necessary to convert workload representations among each other, e.g., transform an event-based arrival curve to a resource-based representation, or vice versa. If each event of a stream imposes the same workload on the corresponding component, this can again be done by simply scaling the arrival curves by an appropriate constant factor (see Scaler component below). If the imposed workload is not homogenous (because of, e.g., different payloads, caching effects), more advanced workload conversion techniques can be applied, see [Wan06, Ch.4] and [Mak06]. In this thesis we assume that all workload conversions are implicit, that is, they are performed whenever needed, but are not shown in the MPA models. In particular, we will use the letter α to denote both event-based and resource-based arrival curves. Only in situations where we want to explicitly distinguish the representations, we will denote event-based arrival curves with $\bar{\alpha}$ and resource-based arrival curves with α.

For the sake of readability, we will also often omit the word 'abstract' when referring to abstract event streams. This means that the term *event stream* will generally stand for a set of possible event traces. In order to avoid confusion, concrete event streams will always be denoted as *event traces* or simply *traces*.

2.2.3.2 Resource model

Similarly to abstract event streams, RTC employs *abstract resource streams* to characterize all possible availability patterns of a processing or communication resource. An abstract resource stream is specified by a tuple of service curves $\beta(\Delta) := [\beta^u(\Delta), \beta^l(\Delta)]$ where $\beta^u(\Delta)$ denotes the upper service curve and $\beta^l(\Delta)$ the lower service curve of the stream.

Def. 5: **(Service curves)** *Let $c(s,t)$ be the service function of any concrete resource availability pattern belonging to an abstract resource stream. Then, c, β^u, β^l are related to each other by the inequality*

$$\beta^l(t-s) \le c(s,t) \le \beta^u(t-s) \quad \forall s \le t \tag{2.15}$$

with $beta^l(0) = \beta^u(0) = 0$. If the above inequality holds for a concrete resource availability pattern c, we say that c conforms to *β, denoted as $c \models \beta$.*

Informally, an upper service curve $\beta^u(\Delta)$ specifies the maximum amount of resource units that are available in *any* time interval of length Δ. Similarly, a lower service curve $\beta^l(\Delta)$ specifies the minimum amount of resource units for any time interval of length Δ. Both upper and lower service curves are monotonically increasing functions. An example of a tuple of service curves that models an abstract resource stream is shown in Figure 9.

Def. 6: **(Set of conforming resource patterns)** *Let β be a tuple of service curves as defined above. The set of all resource patterns that conform to β is defined by*

$$C^\beta := \{c \in C : c \models \beta\}, \tag{2.16}$$

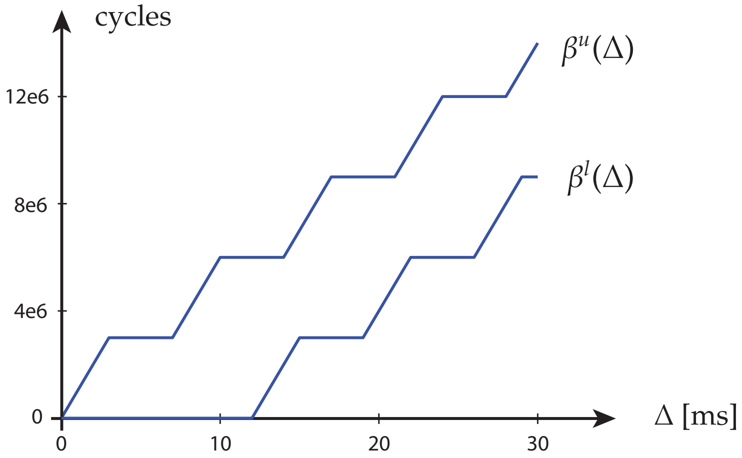


Fig. 9: Service curves (example)

where C denotes the set of all resource patterns.

The conformance of a resource pattern c to an upper (lower) service curve β^u (β^l), as well as the sets C^{β^u}, C^{β^l} are defined accordingly. For a particular resource pattern specified by a service function $c(s,t)$, the tightest service curves β^u_c, β^l_c that model the pattern are given by:

$$\beta^u_c(\Delta) = \sup_{s\in\mathbb{R}} c(s, s+\Delta) \qquad \forall\Delta \geq 0 \tag{2.17}$$

$$\beta^l_c(\Delta) = \inf_{s\in\mathbb{R}} c(s, s+\Delta) \qquad \forall\Delta \geq 0 \tag{2.18}$$

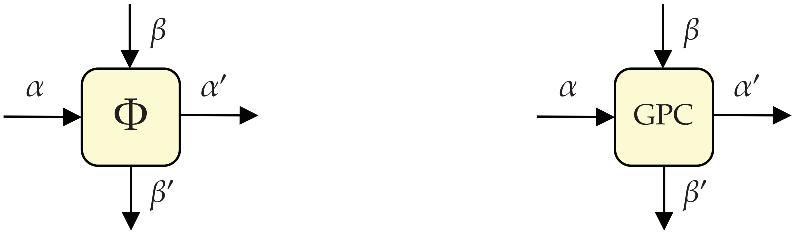
Similarly as for event streams, we will often omit the word 'abstract' when referring to abstract resource streams. This means that the term *resource stream* will generally denote a set of concrete resource availability patterns. Moreover, we will use the term *RTC curves* if we want to generally refer to both, arrival and service curves.

2.2.3.3 Obtaining arrival and service curves

To guarantee the correctness of a performance model, we must ensure that the employed arrival and service curves are exhaustive with respect to system behaviour. Especially when modelling systems with stringent performance requirements (e.g. hard real-time systems), it is essential to cover all concrete instances of event and resource streams observable in the real system. There are several ways to determine correct arrival and service curves for the inputs of a system. One approach is to construct the curves based on some (formal) specification of the corresponding subsystem. For instance, a data sheet could be consulted to derive bounds on the behaviour of some hardware component (e.g. maximum sampling rate of a sensor). Similarly, a description of the physical environment might yield constraints on the workload imposed on the system. A second alternative is to profile a set of representative input event traces or resource patterns, that is, find a (minimal) set of traces such that no other trace of the stream can be 'better' or 'worse'. To identify such representative traces in a stream is, however, often difficult. In the context of this work, we assume that the input arrival and service curves are given, and that their correctness has been verified a priori.

2.2.3.4 Abstract component models

RTC employs *abstract performance components* to model the individual hardware/software units of a system. They operate on abstract event and resource streams, and can be seen as a generalization of the concrete performance components discussed in Section 2.2.2. Figure 10(a) represents a general abstract performance component. The component receives an abstract event stream and an abstract resource stream as input. These streams are specified by a tuple of arrival curves $\alpha = [\alpha^u, \alpha^l]$ and a tuple of service curves $\beta = [\beta^u, \beta^l]$, respectively. They represent the variability of the ingoing workload and the available resources. At its output ports, the abstract performance component again produces an abstract event stream and an abstract resource stream. These streams are described by the tuple $\alpha' = [\alpha'^u, \alpha'^l]$ and the tuple $\beta' = [\beta'^u, \beta'^l]$, respectively. They model the variability of the outgoing workload and the remaining resources. The behaviour of an abstract performance component is characterized by a transfer function $(\alpha', \beta') = \Phi(\alpha, \beta)$. The function Φ reflects the particular processing semantics of the modelled component, and is different for different types of components.



(a) A general abstract performance component

(b) Abstract Greedy Processing Component

Fig. 10: Abstract performance components

We can now define the conformance of a concrete performance component (cf. Figure 5(a)) with respect to an abstract performance component.

Def. 7: **(Conformance of concrete and abstract component)** *Let φ be a concrete performance component with transfer function $(r', c') = \varphi(r, c)$. Let Φ be an abstract performance component with transfer function $(\alpha', \beta') = \Phi(\alpha, \beta)$. Then, φ conforms to Φ (written as $\varphi \models \Phi$) if for all $\alpha, \beta, \alpha', \beta', r, r', c, c'$ the following proposition holds:*

$$\Big[\big(r \models \alpha\big) \wedge \big(c \models \beta\big) \wedge \big((r', c') = \varphi(r, c)\big) \wedge \big((\alpha', \beta') = \Phi(\alpha, \beta)\big)\Big] \Rightarrow \Big[\big(r' \models \alpha'\big) \wedge \big(c' \models \beta'\big)\Big] \tag{2.19}$$

In simpler words, we say that φ conforms to Φ if for any input that conforms to some abstract input streams α, β the component φ produces an output that conforms to the corresponding abstract output streams α', β'.

The abstract variant of the *Greedy Processing Component (GPC)* discussed in Section 2.2.2 is shown in Figure 10(b). Its transfer function is characterized by the following equations [CKT03]:

$$\alpha'^{u} = \min\{(\alpha^u \otimes \beta^u) \oslash \beta^l, \beta^u\} \tag{2.20}$$

$$\alpha'^{l} = \min\{(\alpha^l \oslash \beta^u) \otimes \beta^l, \beta^l\} \tag{2.21}$$

$$\beta'^{u} = (\beta^u - \alpha^l)\, \overline{\oslash}\, 0 \tag{2.22}$$

$$\beta'^{l} = (\beta^l - \alpha^u)\, \overline{\otimes}\, 0 \tag{2.23}$$

The above equations use convolution and deconvolution operators of min-plus and max-plus algebra which are defined as follows:

$$(f \otimes g)(\Delta) = \inf_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \tag{2.24}$$

$$(f \oslash g)(\Delta) = \sup_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} \tag{2.25}$$

$$(f\, \overline{\otimes}\, g)(\Delta) = \sup_{0 \le \lambda \le \Delta} \{f(\Delta - \lambda) + g(\lambda)\} \tag{2.26}$$

$$(f\, \overline{\oslash}\, g)(\Delta) = \inf_{\lambda \ge 0} \{f(\Delta + \lambda) - g(\lambda)\} \tag{2.27}$$

The proofs for Equations (2.20)-(2.23) can be found in [Wan06]. An important observation is that these equations are correct for event and resource traces with unbounded past, i.e., they are used to constrain arrival functions r and service functions c defined on the domain $\mathbb{R} \times \mathbb{R}$. However, if an initial time point is considered for the system evolution, Equations (2.21) and (2.22) are not correct and must be replaced by:[4]

$$\alpha'^{l} = \alpha^l \otimes \beta^l \tag{2.28}$$

$$\beta'^{u} = (\beta^u - \alpha^l)\, \overline{\otimes}\, 0 \tag{2.29}$$

---

[4] Without loss of generality, the initial time point can be considered to be 0. In this case r and c are defined on the domain $\mathbb{R}^{\ge 0} \times \mathbb{R}^{\ge 0}$.

Besides the GPC component, the RTC formalism offers also abstract models for other processing components. Examples are components which serve multiple event streams according to different scheduling policies such as EDF, FIFO or WFQ (see Section 2.2.3.5).

Finally, let us revisit the elementary components considered in Section 2.2.2. RTC offers abstract variants of those components which are shown in Figure 11. They operate on abstract event streams and are characterized by the following transfer functions:

- *Scaler:*

$$\alpha'^{u}(\Delta) = w \cdot \alpha^{u}(\Delta) \tag{2.30}$$

$$\alpha'^{l}(\Delta) = w \cdot \alpha^{l}(\Delta), \tag{2.31}$$

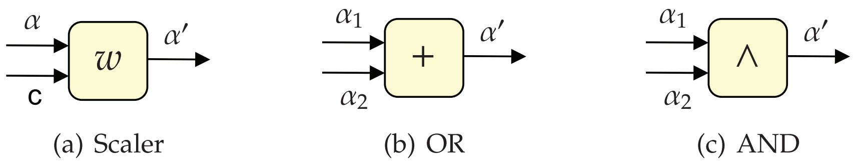
  where w is a positive scaling factor.

- *OR:*

$$\alpha'^{u}(\Delta) = \alpha_1^{u}(\Delta) + \alpha_2^{u}(\Delta) \tag{2.32}$$

$$\alpha'^{l}(\Delta) = \alpha_1^{l}(\Delta) + \alpha_2^{l}(\Delta) \tag{2.33}$$

- *AND:* The transfer functions for the abstract AND component are more involved and can be found in [Wan06, Ch.3].



(a) Scaler (b) OR (c) AND

Fig. 11: Abstract elementary components

2.2.3.5 Representing resource sharing schemes

In Section 2.2.1 we have seen how individual performance components can be composed to form an MPA model for a distributed system. The resulting network of performance components has to reproduce two main aspects of the system:

1. The dataflow among the various system components
2. The hardware architecture of the system, including the assignment of tasks to hardware resources and the adopted resource sharing schemes

The first point can be easily handled by properly connecting the event stream ports of the individual performance components. For instance, the dataflow of the system shown in Figure 3 translates directly to the forwarding of event streams shown in the MPA model of Figure 4. In terms of RTC, these connections signify that the output arrival curves of component *T1* are used as input arrival curves for component *C1*, the output arrival curves of *C1* become the input arrival curves of *T3* and so on.

The second point is, however, less trivial, as not all resource sharing schemes can be modelled by appropriately forwarding resource streams. While for the system of Figure 3 this approach works well, there are also scheduling policies that require the use of dedicated performance components. We will now illustrate the two variants by discussing the RTC models of two particular resource-sharing schemes.

- ***Preemptive Fixed Priority Scheduling***
  Consider a processor that executes n tasks $\tau_1, ..., \tau_n$ with fixed and pairwise different priorities. Assume that the processor schedules the tasks in a preemptive fashion, i.e., an executing low priority task is immediately preempted if a task with higher priority needs to be executed. Without loss of generality, assume that the tasks are ordered by their priorities, i.e., τ_i has higher priority than τ_j if $i < j$. Then, in the formalism of RTC, the processor can be represented by a chain of abstract GPC components as shown in Figure 12. The model has the following inputs and outputs: α_1, ..., α_n specify the abstract event streams that trigger the individual tasks; β models the available processing resources; α'_1, ..., α'_n bound the streams of processed events; β' represents the processing resources left over by the tasks. The propagation of the service curves in the model of Figure 12 naturally reflects the priorities of the tasks: A task τ_i with $i > 1$ gets only the processing resources left over by the task $\tau_i - 1$, which corresponds exactly to the behaviour of a preemptive fixed priority scheduler.

- ***FIFO Scheduling***
  Assume now that the processor schedules the n tasks $\tau_1, ..., \tau_n$ in a first-come, first-served order. This policy is also denoted FIFO scheduling, which stands for first-in, first-out scheduling. In this case, the sharing scheme for the processor cannot be modelled in RTC by properly connecting individual GPC components. Instead, we need to design the tailored abstract processing component shown in Figure 13(a). The inputs and outputs have the same interpretation as in the preemptive fixed priority model. Let us

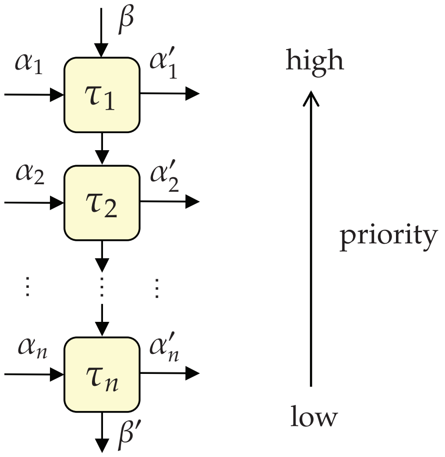


Fig. 12: RTC model for preemptive fixed priority scheduling

now derive a conservative transfer function for the abstract FIFO component by applying simple best-case/worst-case reasonings.

For the computation of the remaining service β' we adopt the following variant of the relations (2.22) and (2.23):

$$\beta'^u = (\beta^u - \sum_i \alpha_i^l)\,\overline{\oslash}\,0 \tag{2.34}$$

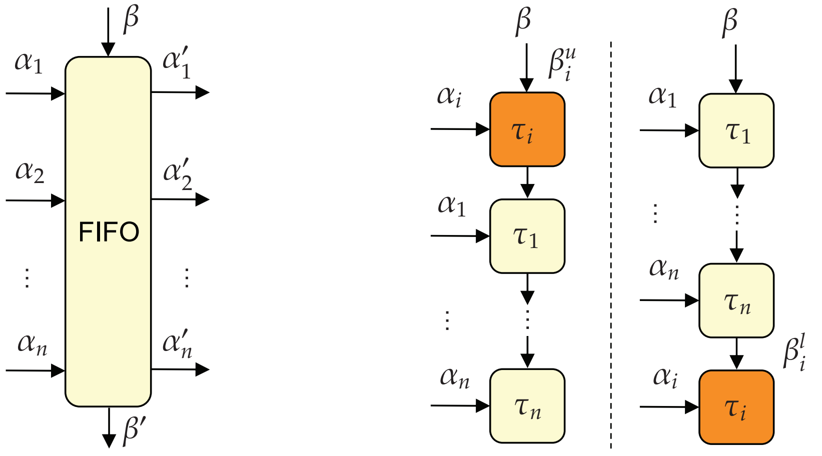
$$\beta'^l = (\beta^l - \sum_i \alpha_i^u)\,\overline{\otimes}\,0 \tag{2.35}$$

Here, we assume feeding a single GPC component with an event stream α which is the sum of all the input event streams $\alpha_1, \cdots, \alpha_n$. In terms of remaining resources, this is clearly equivalent to the case where multiple tasks are triggered by the individual event streams and executed in some given order. To determine valid upper and lower bounds for the individual outgoing event streams, we first look at the maximum and minimum possible amount of processing resources which are available for an individual input event stream. More specifically, for the task associated with input stream α_i we consider the best case and the worst case in a fixed assignment of priorities, as shown in Figure 13(b). The corresponding service curves are given by:

$$\beta_i^u = \beta^u \tag{2.36}$$

$$\beta_i^l = (\beta^l - \sum_{j \neq i} \alpha_j^u)\,\overline{\otimes}\,0 \tag{2.37}$$

Under FIFO scheduling, the amount of resources available for the processing of α_i can obviously only be less or equal than β_i^u and larger



(a) Abstract FIFO component

(b) Best-case and worst-case scenario for event stream α_i under fixed priorities

Fig. 13: Modelling FIFO scheduling in RTC

or equal than β_i^l. Hence, the two bounds are a valid abstraction. Given β_i^u and β_i^l, we can compute bounds for the corresponding outgoing event stream α'_i by using the relations (2.20) and (2.21):

$$\alpha_i'^{u} = \min\{(\alpha_i^u \otimes \beta_i^u) \oslash \beta_i^l, \beta_i^u\} \tag{2.38}$$

$$\alpha_i'^{l} = \min\{(\alpha_i^l \oslash \beta_i^u) \otimes \beta_i^l, \beta_i^l\} \tag{2.39}$$

The RTC formalism also provides models for several other scheduling policies. For instance abstract performance components analogous to the one shown in Figure 13(a) exist for Earliest Deadline First (EDF) scheduling [WT06a], non-preemptive fixed priority scheduling [HT07], and Weighted Fair Queue (WFQ) scheduling.

A main advantage of the explicit resource characterization in MPA is that it makes the method compositional also in terms of resource sharing schemes. In particular, we can combine two or more of the above performance components to model hierarchical scheduling policies. As an example, consider the MPA model shown in Figure 14. It represents a hierarchical scheduling scheme, in which two tasks share a processor according to the preemptive fixed priority policy, but the second task is actually processing two event streams in FIFO order.

2.2.3.6 Performance analysis

The purpose of building an MPA model is to quantify the performance of a system. Various performance metrics such as event latencies or backlogs can be relevant for a system designer. The RTC formalism can derive hard

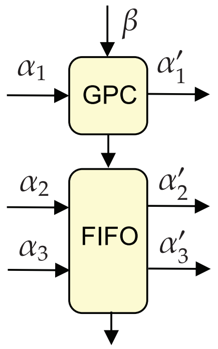


Fig. 14: Hierarchical scheduling

bounds for such system characteristics. More specifically, RTC permits us to analytically quantify the best-case and worst-case performance of each abstract performance component as a function of the input arrival and service curves. In the following, we present the analytical expressions for the worst-case performance bounds of an abstract GPC component. Note that the expressions can also be directly used for all the abstract performance components that use GPC as a building block. For instance, this is the case for the FIFO component discussed above.

For the maximum buffer fill level (or backlog) b_{max} of a GPC component with activation pattern $\alpha = [\alpha^u, \alpha^l]$ and resource availability $\beta = [\beta^u, \beta^l]$ we have $b_{max} \leq Buf(\alpha^u, \beta^l)$ where the bound $Buf(\alpha^u, \beta^l)$ is defined as

$$Buf(\alpha^u, \beta^l) = \sup_{\lambda \geq 0}\{\alpha^u(\lambda) - \beta^l(\lambda)\}. \tag{2.40}$$

For the maximum delay d_{max} experienced by an input event at the same GPC component we have $d_{max} \leq Del(\alpha^u, \beta^l)$ where the bound $Del(\alpha^u, \beta^l)$ is defined as

$$Del(\alpha^u, \beta^l) = \sup_{\lambda \geq 0}\Big\{\inf\{\tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau)\}\Big\}. \tag{2.41}$$

Figure 15 shows the graphical interpretation of these two quantities. The bound for the worst-case backlog corresponds to maximum vertical distance between α^u and β^l. On the other hand, the bound for the worst-case delay corresponds to the maximum horizontal distance between these two curves.

As already indicated in Section 2.2.2, we can derive system-wide performance characteristics by aggregating the results obtained for individual components. For instance, to quantify the worst-case end-to-end delay for an event stream that traverses several components, we can first bound the worst-case delay observable at each component by means of (2.41), and then simply compute the sum of those delays.

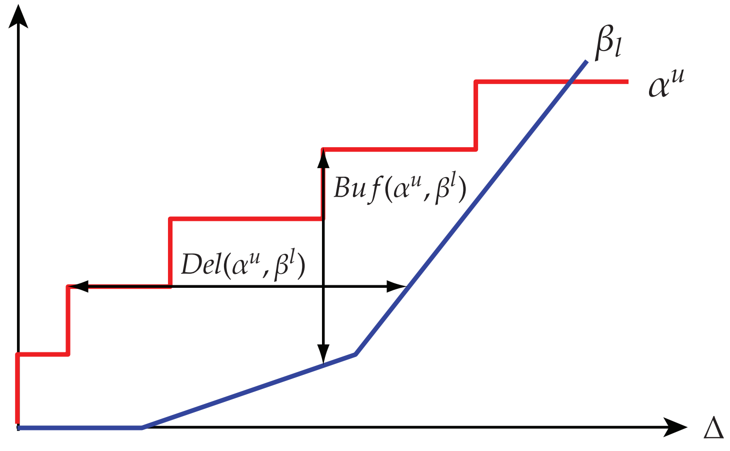


Fig. 15: Graphical interpretation of $Del(\alpha^u, \beta^l)$ and $Buf(\alpha^u, \beta^l)$

Summing up individual worst-case performance bounds can, however, lead to very conservative results. The reason for this is that in a sequence of tasks, the worst-case event burst can often not appear in succession at every single task. In the literature, this phenomenon is denoted as 'Pay burst only once' [LT01]. For such a sequence of GPC components the bound for the total delay can be tightened to

$$d_{max} \leq Del(\alpha^u, \beta_1^l \otimes \beta_2^l \otimes \ldots \otimes \beta_n^l), \tag{2.42}$$

where α denotes the input event stream processed by the task sequence, and β_1, ... β_n denote the service offered to the single tasks. The same reasoning applies to the computation of the total buffer space needed by a sequence of GPC components.

2.2.3.7 Implementation

The framework for Modular Performance Analysis with Real-Time Calculus has been implemented in the RTC Toolbox for Matlab [WT][5].

The main problem to tackle in an implementation of the RTC formalism is to find an appropriate finite datatype to represent arrival and service curves which are defined on an infinite domain. The chosen representation needs to be precise enough to enable an accurate performance analysis, but at the same time abstract enough to achieve efficiency. In the RTC Toolbox, this issue is tackled by means of a compact representation of RTC curves which relies on a list of piecewise linear segments. The segments are used to specify an aperiodic curve, a periodic curve, or a curve consisting of both an aperiodic and a periodic part. More detailed

---

[5] available at `http://www.mpa.ethz.ch/Rtctoolbox`

information on efficient representation of RTC curves can be found in [Wan06, Ch.7].

2.2.3.8 Case study: Analysis of an industrial embedded system

The MPA framework with RTC has been successfully applied to the analysis of a large-scale industrial embedded system [SPLT10]. The considered system is an avionic in-cabin communication system designed by a major airplane manufacturer. The system, denoted as Heterogeneous Communication System (HCS), is based on Ethernet network and connects different electronic components of an aircraft cabin such as sensors, speakers, video cameras, routers and a central server. Figure 16 illustrates the basic architecture of the HCS. The server is connected to more than 200 end devices over several Network Access Controllers (NAC). The HCS is used by a set of specific applications which produce network traffic. Examples of applications are the synchronization of device clocks, crew announcements, video surveillance and audio streaming. Besides these applications, in the HCS there is also network traffic initiated by user interaction (e.g. service calls) and background traffic (e.g. signalling data). The goal of the analysis is to determine whether the system design meets several requirements, mostly related to its timing behaviour. For instance, the specification defines upper bounds for tolerable clock drifts and end-to-end delays in the HCS.

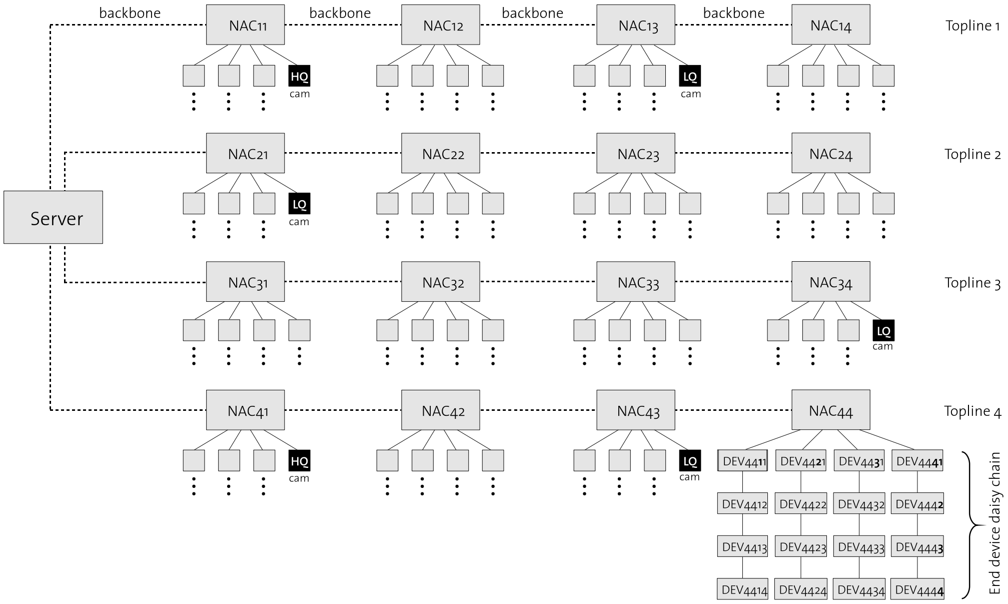


Fig. 16: Architecture of the Heterogeneous Communication System

To verify the above requirements, an MPA model of the HCS was constructed. This involved the characterization of all the traffic sources by means of arrival curves, as well as the modelling of the various system parts by means of MPA components. The MPA model of the HCS contains mostly GPC and FIFO components. For instance, the behaviour of the Ethernet links can be abstracted by means of FIFO components. Besides these components, also dedicated MPA components were employed in the model. In particular, an MPA component for modelling Weighted Fair Queue (WFQ) scheduling was designed.

The major challenge for the MPA analysis of the HCS was given by the size of the system. In principle, arbitrary large systems can be modelled and analyzed with MPA. In practice, however, when analyzing large systems with the RTC Toolbox, one often faces long run-times and large memory demands. The reason for this problem is the detailed representation of arrival and service curves mentioned in Section 2.2.3.7. More specifically, the complexity of the MPA analysis depends on the particular shape of the involved arrival and service curves. In large MPA models, where the input arrival curves are manipulated by long sequences of components, the resulting curves tend to be very complex, i.e. they consist of a large number of segments. In order to make the analysis of the HCS more efficient, a simple but efficient approximation scheme for arrival and service curves was developed. The conservative approximation scheme avoids periodic patterns in the representation of curves which proved to be crucial to achieve fast evaluations of large systems. In the particular case of the HCS, the approximation scheme permitted us to reduce the analysis time by two orders of magnitude with a negligible loss of analysis accuracy.

The complete documentation of the HCS case study including all modelling assumptions and the full list of analysis results can be found in [SPLT10]. The report also contains a description of the approximation scheme proposed to speed up the MPA analysis of large systems.

Finally, we would like to note that the MPA framework is not necessarily bound to RTC for analyzing the performance of a system. Instead of RTC, any other suitable formalism that handles event and resource streams can be used. For instance, in Chapter 5 we will show how to embed the state-based formalism of Timed Automata into the MPA framework. Nevertheless, most of the results presented in this thesis are based on RTC. Hence, from now on, whenever we refer to ’MPA’, we implicitly mean ’MPA with RTC’.

2.3 Evaluation and Comparison of Abstractions for Performance Verification

The formal methods of performance evaluation discussed in Section 2.1 are based on essentially different concepts of abstraction. While all these methods provide hard bounds for the performance characteristics of distributed systems, they often lead to remarkably different bounds, depending on the analyzed system. In other words, they exhibit different kinds of abstraction losses. It is therefore important to clarify what the influence of the individual modelling techniques on the accuracy of the performance evaluation is. To this end, in [Per06] we performed a quantitative comparison of several formalisms for performance evaluation. The comparison is based on a set of benchmark systems that highlights specific pitfalls for the different evaluation techniques. In the following, we reproduce and extend our results of [Per06], as they inspired most of the work presented in this thesis.

2.3.1 Motivation and Challenges

An evaluation of the different formalisms is desirable for several reasons. First of all, such an assessment allows us to highlight the effects of the particular abstractions. It helps to determine the modelling limits and the analysis pitfalls for different methods. And more importantly, the comparison serves to better understand the relation between adopted models and achieved accuracy, as well as to improve the precision of the formalisms by combining ideas and abstractions.

Conducting a direct comparison of different formalisms for performance evaluation is difficult for several reasons. First of all, the various approaches differ substantially in terms of expressivity, modelling effort, and scalability. Also in terms of tool support, the variety is wide; while some formalisms are implemented in powerful commercial software tools, others are available as academic prototypes only, or are not implemented at all. Moreover, several important aspects of the abstractions can only hardly be quantified. For instance, this is the case for properties such as modelling power or scalability. Another fundamental problem is that the modelling scopes of the methods do only partially intersect. This means that a method often allows us to model scenarios that are not covered by other techniques. Hierarchical scheduling, blocking times or complex task activation patterns are just a few examples of numerous system properties that differentiate the modelling power of the various abstractions.

2.3.2 Comparison Methodology

For the comparison, we choose four state-of-the-art tools for performance evaluation of distributed real-time systems:

- MAST (v1.3.6)
  Aggregates and implements several algorithms of holistic scheduling analysis. Developed at the University of Cantabria.
  Available at `http://mast.unican.es/`.
- SymTA/S (v1.1)
  Commercial tool for compositional performance analysis. Developed by Symtavision GmbH (`http://www.symtavision.com/`).
- RTC Toolbox (v1.0)
  Tool for compositional performance analysis based on the MPA framework with Real-Time Calculus.
  Available at `http://www.mpa.ethz.ch/rtctoolbox/`.
- Uppaal (v4.0.3)
  Model checker for state-based analysis of real-time systems based on the formalism of Timed Automata (TA).
  Available at `http://www.uppaal.com`.

Concerning the performance evaluation with Uppaal, we follow the approach proposed by Hendriks and Verhoef [HV06], which is based on dedicated TA representations of PJD models, as well as TA observers for system performance metrics. In addition to the four formal methods for worst-case performance evaluation, we also employ a simple SystemC-based discrete event simulator. This is done in order to illustrate the difference among the performance bounds obtainable by formal and empirical methods.

To enable a direct quantitative comparison of the formalisms, we decide to focus mostly on systems in the intersection of the modelling scopes. This allows us to highlight the specific effects of the different abstractions. We intentionally keep the considered benchmark systems small, with the purpose of isolating the influence of particular system characteristics, and exposing specific analysis difficulties. In order to produce meaningful evaluations, we do not restrict the benchmarks to single system configurations but specify sets of values for relevant system parameters.

Furthermore, we would like to point out that the comparison is not intended as competition between formalisms for performance evaluation. Given the large heterogeneity of the modelling capabilities, such a competition could hardly be fair. Rather than ranking the individual approaches, our main motivation is the detection and investigation of abstraction losses.

2.3.3 Benchmark Systems

In this section, we describe a set of benchmark systems that are used for the evaluation of the different formalisms. Some of the described benchmark systems were discussed, among others, at the ARTIST2 Workshop on Distributed Embedded Systems.[6] Every benchmark is tailored to a particular analysis issue, and consists of a simple system architecture (involving only few event streams, tasks and resources) and a performance characteristic to be quantified. The set of proposed benchmarks is not exhaustive by far, as there are lots of system configurations that lead to challenging analysis problems. However, our benchmarks define several orthogonal scenarios that lead to abstraction losses for different analysis tools. For the sake of simplicity, we define only benchmarks with constant task execution times. This choice is made to permit an easier interpretation of the analysis results but is not strictly necessary, since all the considered tools can handle variable task execution times, typically specified as intervals $[BCET, WCET]$. Moreover, in the described benchmarks the input streams are fully asynchronous and the buffering of events does not affect the performance of the system, i.e. we consider unbounded and infinitely fast buffers.

2.3.3.1 Benchmark 1: Complex task activation pattern

The intention of this benchmark is to compare the ability of different abstractions to handle complex task activation patterns. By *complex* we mean activation patterns that cannot be precisely captured by means of a PJD model.

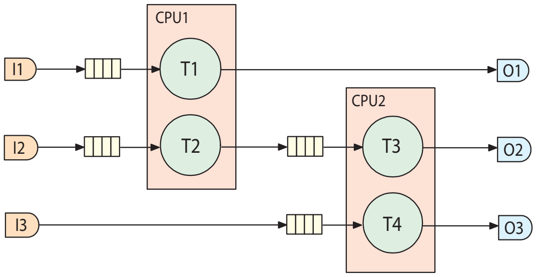
Figure 17 depicts the architecture of the benchmark system. Three periodic event streams are processed by four tasks running on two CPUs that implement preemptive fixed priority scheduling. The performance characteristic to determine is the worst-case response time of task *T3* as a function of the period of stream *I3*. Note that the activation pattern of task *T3* is not periodic anymore, since task *T1* can preempt task *T2*. In other words, the output behaviour of task *T2* is complex, i.e., cannot be precisely described by a PJD model. Thus, we expect pessimistic analysis results for abstractions relying on PJD models.

2.3.3.2 Benchmark 2: Variable feedback

The purpose of this benchmark is to confront the different formalisms with a feedback loop, and the consequent correlations among the activation times of the involved tasks. The system topology is shown in Figure 18. A periodic event stream *I2* is processed in sequence by three

---

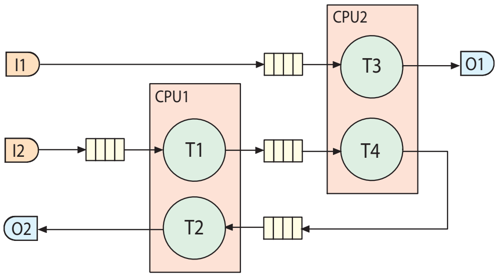
[6] `http://www.tik.ee.ethz.ch/~leiden05/`



Input streams	I1: periodic (P = 60ms) I2: periodic (P = 5ms) I3: periodic (P = [60..110]ms)
Resource sharing	CPU1: FP preemptive, CPU2: FP preemptive
Execution times	T1: 35ms, T2: 2ms, T3: 4ms, T4: 12ms
Scheduling parameters	priority T1: high, priority T2: low priority T3: low, priority T4: high

Fig. 17: Specification of Benchmark 1

tasks running on two CPUs and forming a feedback loop. In addition, *CPU2* also processes a second periodic event stream *I1* of higher priority. The performance metric to determine is the worst-case latency from *I2* to *O2*. In order to vary the correlation among the task activation times in the feedback loop, the benchmark specifies different values for the execution time of task *T3*. Since the compositional formalisms do not take into account such correlations among task activation times, we expect abstraction losses for these methods.

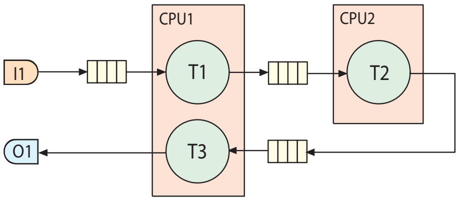


Input streams	I1: periodic (P = 100ms) I2: periodic (P = 5ms)
Resource sharing	CPU1: FP preemptive, CPU2: FP preemptive
Execution times	T1: 2ms, T2: 2ms, T3: [2..22]ms, T4: 1ms
Scheduling parameters	priority T1: high, priority T2: low priority T3: high, priority T4: low

Fig. 18: Specification of Benchmark 2

2.3.3.3 Benchmark 3: Cyclic dependencies

The intention of this benchmark is to examine the capability of the different abstractions to deal with cyclic dependencies.



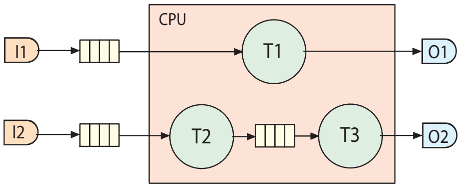
Input stream I1	periodic with burst (P = 10ms, J = [0..50]ms)
Resource sharing	CPU1: FP preemptive
Execution times	T1: 1ms, T2: 4ms, T3: 4ms
Scheduling parameters	1) priority T1: high, priority T3: low 2) priority T1: low, priority T3: high

Fig. 19: Specification of Benchmark 3

Figure 19 represents the system to evaluate. A periodic event stream with bursts is processed by a sequence of three tasks running on two resources. On *CPU1* a preemptive fixed priority scheduler is used to execute the tasks *T1* and *T3*. The performance characteristic to determine is the worst-case delay from *I1* to *O1* for increasing values of the input jitter and in two different scenarios. In scenario 1, *T1* has higher priority than *T3*. In scenario 2, *T3* has higher priority than *T1*, i.e., there is a cyclic dependency between the two tasks: *T1* indirectly triggers *T3*, but *T3* preempts *T1*). We expect this cyclic dependency to make the evaluation difficult for compositional system abstractions.

2.3.3.4 Benchmark 4: Data dependencies

The purpose of this benchmark is to quantify the abstraction losses experienced with different formalisms for the analysis of systems with data dependencies among tasks. The system specified below was first presented as example in [YW95] by Yen and Wolf. Figure 20 depicts the architecture of the system. Two periodic event streams are processed by three tasks on a CPU that implements preemptive fixed priority scheduling. The data dependency is given by the execution sequence *T2*-*T3*. The performance metric to determine is the worst-case delay from *I2* to *O2* as function of the execution time of *T1*. For this benchmark we expect pessimistic analysis results for abstractions that cannot take into consideration data dependencies among tasks.

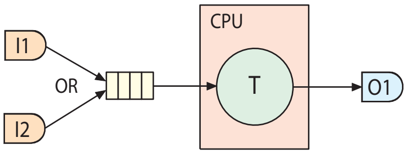


Input streams	I1: periodic (P = 80ms) I2: periodic (P = 50ms)
Resource sharing	CPU: FP preemptive
Execution times	T1: [15..30]ms, T2: 20ms, T3: 10ms
Scheduling parameters	priority T1: high, priority T2: medium, priority T3: low

Fig. 20: Specification of Benchmark 4

2.3.3.5 Benchmark 5: Multiple inputs with OR-activation

The intention of this benchmark is to compare the different abstractions with respect to the combination of multiple event streams for the activation of tasks. We consider a simple system consisting of a task with two inputs in OR-combination, i.e., each event on both input streams activates the task. The system topology is shown in Figure 21.



Input streams	I1: periodic with jitter (P = 100ms, J = 20ms) I2: periodic with jitter (P = 150ms, J = 60ms)
Execution time	T: [25..60]ms

Fig. 21: Specification of Benchmark 5

Task *T* is triggered by the events of two periodic streams with jitter which queue up in a shared FIFO buffer. The performance characteristic to determine is the worst-case delay from *I1* to *O1* as function of the execution time of task *T*. Since the sum of the two input event streams cannot be accurately represented with a PJD model, we expect loose performance bounds for formalisms relying on these models.

2.3.4 Analysis Results

In this section, we present the results obtained by applying the formal performance evaluation methods listed in Section 2.3.2 to the benchmarks of Section 2.3.3. We compare the performance bounds obtained and discuss differing results and analysis pitfalls. All the models adopted are available online.[7]

In the following graphs, the performance results obtained by discrete event simulation are shown for comparison only. They illustrate that empirical methods for performance evaluation can sometimes miss corner cases of the system behaviour. Details on the employed simulator, PESIMDES [Per], can be found online.[8] The length of the simulated traces is shown in brackets in the graphs.

2.3.4.1 Benchmark 1: Complex task activation pattern

In this experiment we evaluate the accuracy of the different formalisms when some event patterns in the system deviate from PJD event models. For this purpose, we tap the event stream between the tasks *T2* and *T3* in the system of Figure 17, where a distortion of the periodic event pattern occurs due to the influence of task *T1*. Figure 22 shows the analysis results for the worst-case response time of task *T3*.[9] The performance values derived with Uppaal are obtained through model checking, and represent the exact worst-case response time of task *T3*. In other words, there is no abstraction loss for the results determined by Uppaal. In general, this statement holds only if one can show that the TA model accurately reflects the behaviour of the real system. For the considered benchmark systems this is, however, trivial. As a result, we can use the Uppaal values as reference for the assessment of the other formalisms.

The graph shows that the compositional methods provide pessimistic predictions for the worst-case response time of task *T3*. It also points out that there is a remarkable difference between the results obtained by SymTA/S and MPA. This can be explained by the different event models employed by the two abstractions. While MPA accurately models the complex output pattern of *T2* by an appropriate pair of arrival curves, SymTA/S approximates the output of *T2* by a periodic event stream with burst.

Figure 23 shows the effect of the two different event models on the analysis accuracy for the case $P_{I3} = 65$ ms. The worst-case response time of *T3* is given by the maximum horizontal distance between the worst-case resource availability (dashed curve), and the worst-case execution demand (solid curves). The graph illustrates that the approximation adopted by SymTA/S leads to an overestimated response time of task *T3*. Interestingly, however, the pessimistic event model of SymTA/S has

---

[7] `http://www.tik.ee.ethz.ch/~leiden05/index2.html#publications`

[8] `http://www.mpa.ethz.ch/PESIMDES/Overview`

[9] The MAST tool does not support the analysis of local response times in the current release and was thus not considered for this analysis problem.

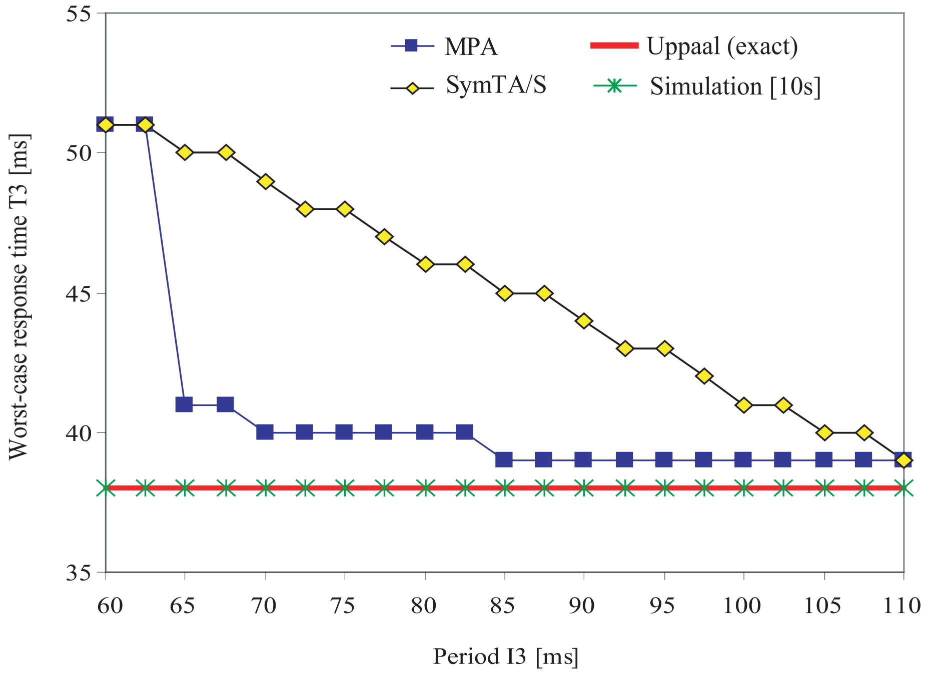


Fig. 22: Analysis results for the WCRT of *T3* in Benchmark 1

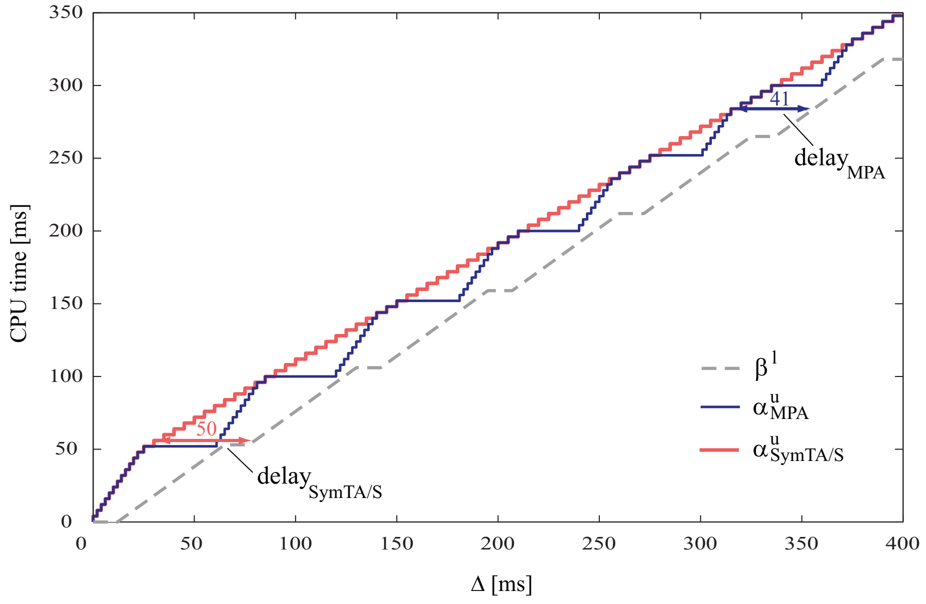


Fig. 23: Influence of different event models on the WCRT analysis of *T3* for P_{I3} = 65 ms

no negative effect on the total delay from *I2* to *O2*. This is because the adopted path analysis detects that the total worst-case delay from *I2* to *O2* is smaller than the sum of the two single worst-case delays ('Pay burst only once' phenomenon). For the worst-case delay *I2-O2* all considered formalisms determine the exact performance results.

2.3.4.2 Benchmark 2: Variable feedback

In Benchmark 2, depicted in Figure 18, the behaviour of the feedback stream *I2-O2* depends strongly on the execution time of task *T3*. The reason is that task *T3* may preempt task *T4* and thus affect its response time. In particular, increasing the worst-case execution time of *T3* in uniform steps causes the correlation effects between *T1* and *T2* to oscillate in a periodic manner. This effect is shown in Figure 24 by the exact values of the worst-case delay *I2-O2* determined with Uppaal. The graph shows that also the MAST tool provides the exact worst-case performance values for all the parameter configurations of the specified benchmark. However, the compositional analysis approaches MPA and SymTA/S do recurrently overestimate the performance of the system, and provide pessimistic predictions for several parameter values. A closer analysis of the behaviour of the feedback loop reveals that this overestimation happens for those parameter configurations that lead to the worst-case delay *I2-O2*, without a full preemption of task *T2* on *CPU1*. Since the compositional abstractions cannot take into consideration the correlation between the activation times of *T1* and *T2*, the corresponding evaluation methods have no means of recognizing the missing or partial preemption. Hence, they assume that a full preemption is possible in the worst-case, which leads to pessimistic performance bounds.

2.3.4.3 Benchmark 3: Cyclic dependencies

In the first scenario of the specification depicted in Figure 19, task *T1* has higher priority than task *T3*, and thus there is no cyclic dependency in the system. However, correlation effects as described for Benchmark 2 are present. For instance, depending on the input stream parameters, it may happen that task *T3* is not preempted by task *T1*. Such correlations are not fully exploited by all formalisms, as already mentioned above. While the Uppaal model permits us to determine the exact worst-case latencies of the system, other formal evaluation methods like MPA and SymTA/S slightly exceed the exact performance values, and their pessimism grows with increasing input jitter, as shown in Figure 25.

The poor performance predictions of MAST have another cause. Holistic analysis methods compute the worst-case delay not referring

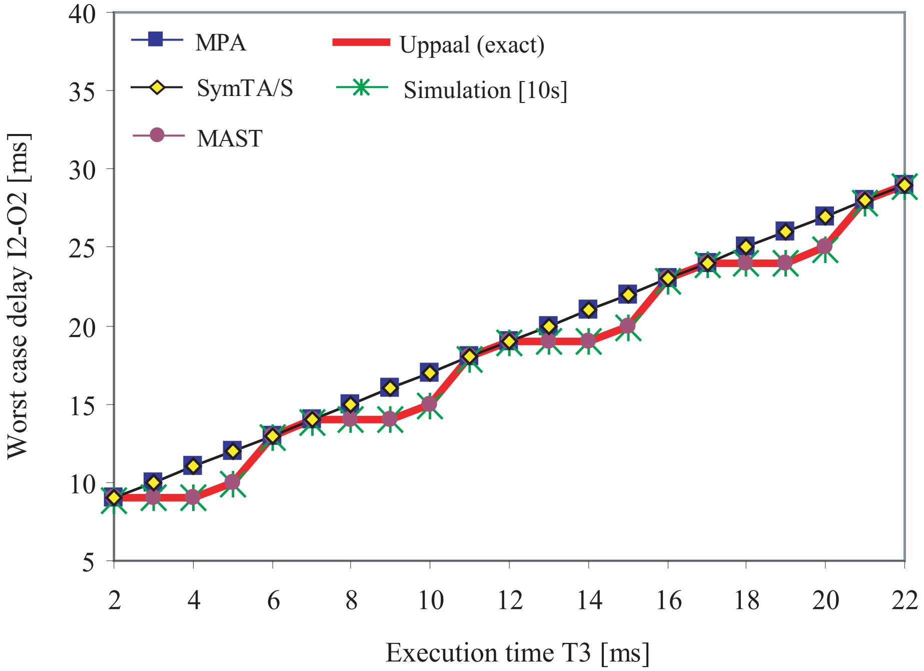


Fig. 24: Analysis results for the worst-case delay *I2-O2* in Benchmark 2

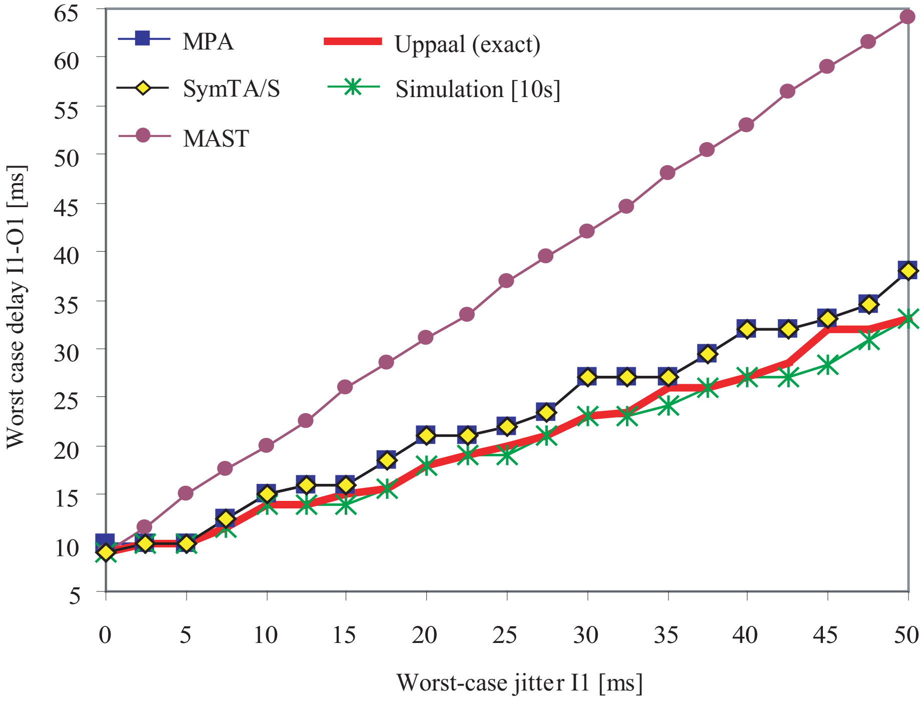


Fig. 25: Analysis results for the worst case delay *I1-O1* in Benchmark 3 (scenario 1)

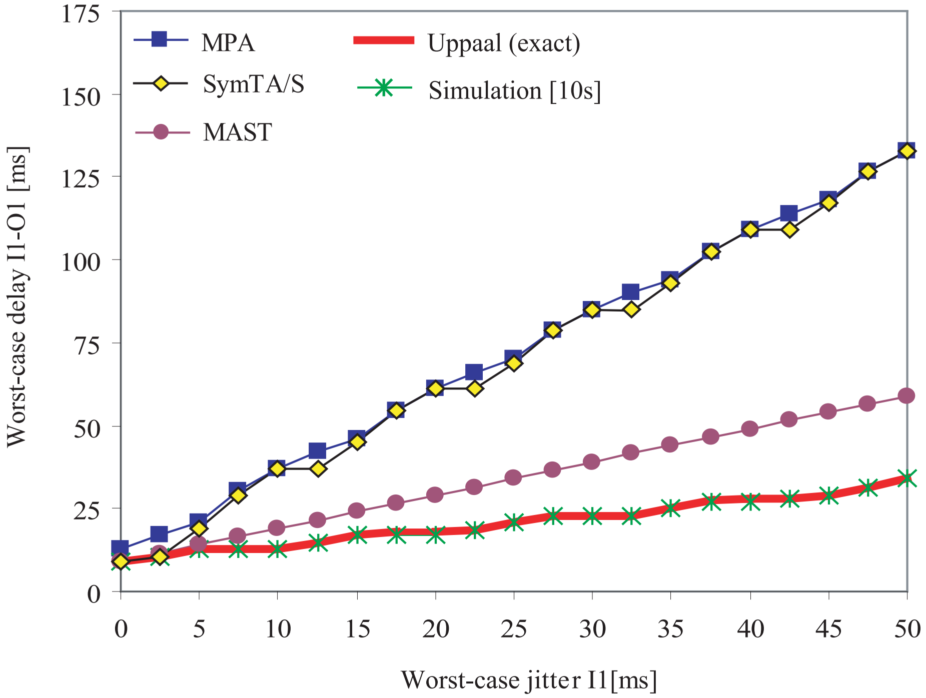


Fig. 26: Analysis results for the worst case delay *I1-O1* in Benchmark 3 (scenario 2)

to the actual release time of an event, which varies within the jitter interval, but referring to the ideal periodic release time. In other words, the release jitter is considered part of the delay already, and thus the predicted worst-case end-to-end latency cannot be smaller than the maximum input jitter. This explains why the pessimism of the predictions provided by MAST increases for increasing values of the input jitter. Unfortunately, this deviation with respect to the other formalisms cannot simply be adjusted after the analysis. The reason is that the actual release instant leading to the worst-case performance is generally unknown. However, the interpretation of latency adopted by MAST is useful in other settings. For instance, in a system in which the activation jitter of a task is caused by a low resolution clock, it is more appropriate to refer the deadline for the response time of the task to the real activation request rather than to the actual activation instant.

The graph in Figure 25 also shows that simulation can in general not be used to guarantee hard performance bounds. In fact, for some input configurations, the corner-cases with worst-case performance are missed by the simulator.

In scenario 2, *T3* has higher priority than *T1* and thus there is a cyclic dependency: The output behaviour of *T1* depends on the CPU availability left over by *T3*, while at the same time, the activity of *T3* depends on the output behaviour of *T1*. For the holistic system abstraction this dependency does not make the analysis more difficult. Figure 26 shows that the performance predictions provided by MAST do not differ more significantly from the exact values than in the previous scenario (note the different scaling of the ordinate axes). However, for compositional abstractions the cyclic dependency complicates the analysis process. Both MPA and SymTA/S use a fixpoint calculation to handle it, but the graph shows that this leads to overly pessimistic performance predictions.

2.3.4.4 Benchmark 4: Data dependencies

Figure 27 displays the performance results obtained by applying the different abstractions to Benchmark 4 specified in Figure 20. The chart shows that MPA and SymTA/S largely overestimate the worst-case delay *I2-O2*, while MAST determines the exact worst-case performance of the system. The overly pessimistic performance prediction of the former two approaches results from the disregard of data dependencies in the system. In particular, the activation times of the tasks *T2* and *T3* are not independent. The data dependency forces the two tasks to be executed in a fixed order, and imposes a temporal offset between their activation.

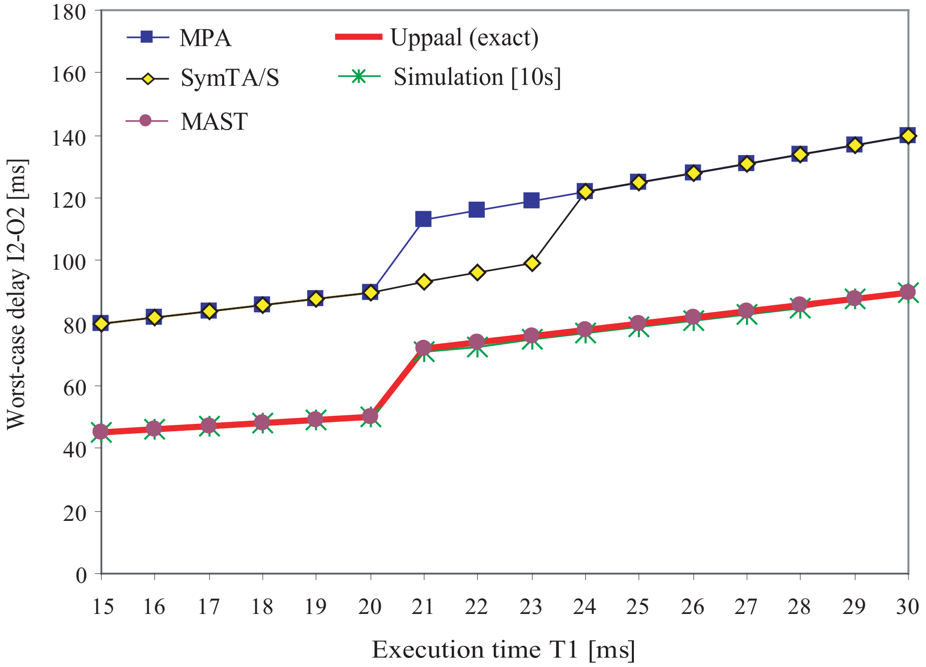


Fig. 27: Analysis results for the worst case delay *I2-O2* in Benchmark 4

For instance, let us consider the system configuration with an execution time of 15 ms for *T1*. It is simple to verify that in this configuration *T1* can only preempt either *T2* or *T3*, but not both in a single execution. Moreover, *T2* cannot preempt *T3*. Hence, the worst-case latency *I2-O2* is 45 ms. However, MPA and SymTA/S ignore the data dependency between *T2* and *T3*. They consider the activation times of the two tasks as completely independent. Thus, they suppose a worst-case response time of 35 ms for *T1* and 45 ms for *T2*. As result, they predict a worst-case latency of 80 ms for the path *I2-O2*, which corresponds to the sum of the two individual delays. In contrast, the MAST tool implements offset-based analysis methods, designed for detecting and exploiting data dependencies among tasks. This permits us to determine tighter performance bounds.

2.3.4.5 Benchmark 5: Multiple inputs with OR-activation

As for the previous benchmarks, the exact performance results have been determined with Uppaal. In this case it was, however, necessary to extend the TA models proposed in [HV06]. The issue is that the original modelling approach uses simple counter variables to represent the fill level of event buffers. This implies that in the TA model, all events stored in a buffer are indistinguishable. However, in the present benchmark, it is necessary to distinguish the activations of *T* caused by input events from *I1* and *I2*. In particular, to permit the quantification of the latency *I1-O1*, the model of the shared buffer has to reflect the arrival order of the two different event types. Hence the input buffer can no longer be represented by means of a counter variable and must be modelled explicitly. Figure 28 shows the TA used to model the input buffer. The automaton basically enumerates all possible buffer states. For the particular parameters of the benchmark, the backlog of *T* does not exceed 3 events, and thus the automaton has only 16 locations. However, the number of locations grows exponentially with the maximum backlog: For an OR-activated task with m inputs and a maximum backlog of n activation requests, an automaton with $\frac{m^{n+1}-1}{m-1}+1$ locations is required. Therefore, in general, this way of modelling the OR-activation involves an impractical modelling and verification effort.

Figure 29 shows the analysis results for the worst-case delay *I1-O1* as determined by the different formalisms. The graph shows that the two compositional evaluation approaches provide different results: MPA determines the exact performance of the system for all the considered parameter values, whereas SymTA/S is more conservative for certain parameters. As for Benchmark 1, the difference originates from the different event models adopted by the two abstractions. The combination of the two event streams *I1* and *I2* can be accurately modelled by MPA, whereas it is approximated with a PJD model in SymTA/S. The more pessimistic performance predictions of MAST are again related to the different inter-

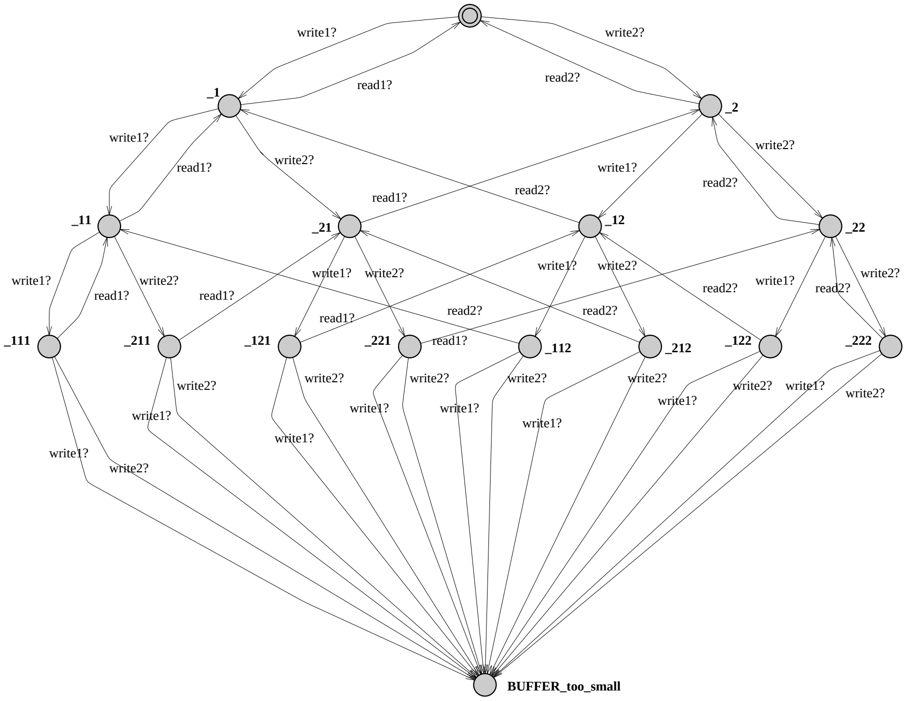


Fig. 28: TA model for the activation buffer in Benchmark 5

pretation of jitter for the input streams. The chart of Figure 29 also shows that the discrete event simulator misses the worst-case performance of the system for several parameter configurations.

2.3.4.6 Analysis times

In the present assessment of performance evaluation methods, we do not only want to quantify the accuracy of the achieved results but also the efficiency of their computation. For this reason, we measured the analysis or verification times of the different tools for all considered benchmarks. Table 2 sums up the minimum, median and maximum analysis time for each performance evaluation tool and benchmark. The values show that most of the considered abstractions permit a fast performance evaluation for all the specified benchmarks. However, the analysis approach based on model checking of timed automata networks forms an exception, as in two benchmarks it suffers from very long verification times. For instance, in the first scenario of Benchmark 3, the maximum verification time of the Uppaal model checker is more than a hundred times larger than the analysis times of the other tools. Especially in the presence of large jitters the state space of the TA models grows considerably, and leads to long verification times.

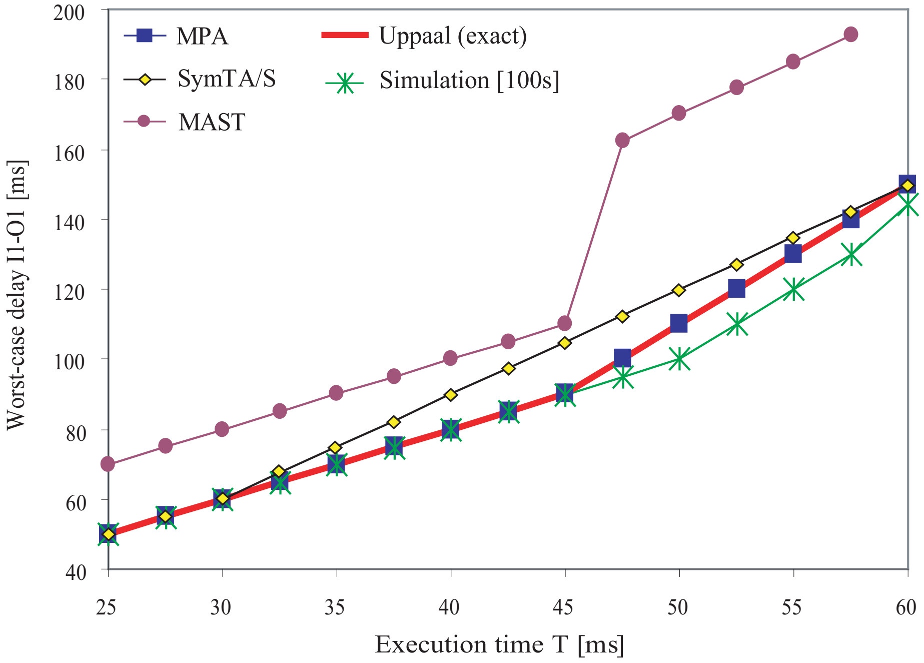


Fig. 29: Analysis results for the worst-case delay *I1-O1* in Benchmark 5

Another interesting observation is that in some cases there is a remarkable difference between the minimum, median and maximum analysis time. This shows that, in general, the analysis times of the different approaches may depend highly on the particular system parameters.

2.3.5 Discussion

The results of Section 2.3.4 show that the accuracy of the performance predictions determined with a given formalism varies considerably for the different benchmarks. The only exception is the approach based on Uppaal, which provides the exact performance predictions for all the considered benchmarks. However, the exact results are often paid for by a large analysis effort, i.e., Uppaal may require very long verification times. Thus, considering not only the achieved accuracy, but also the necessary analysis times, we can state that none of the considered abstractions performed best in all the benchmarks.

Nevertheless, the results permit us to give some indications as to which abstractions are more appropriate than others for the evaluation of particular scenarios. For instance, Benchmarks 1 and 5 indicate that the approximation of complex event streams with PJD models can be inappropriate for precise performance predictions at a local level. Benchmark 3 emphasizes that systems with cyclic dependencies represent a serious

		B1	**B2**	**B3(1)**	**B3(2)**	**B4**	**B5**
MPA	min	0.60	0.03	0.01	0.04	0.03	0.01
	med	1.06	0.04	0.01	0.15	0.05	0.01
	max	19.72	0.08	0.04	0.30	0.20	0.05
SymTA/S	min	0.05	0.03	0.03	0.03	0.06	0.01
	med	0.09	0.05	0.06	0.34	0.09	0.01
	max	1.50	0.23	0.09	0.80	0.31	0.01
MAST[a]	min	-	<0.5	<0.5	<0.5	<0.5	<0.5
	med	-	<0.5	<0.5	<0.5	<0.5	<0.5
	max	-	<0.5	<0.5	<0.5	<0.5	<0.5
Uppaal.[a,b]	min	18.0	<0.5	<0.5	<0.5	<0.5	<0.5
	med	34.5	<0.5	1.0	<0.5	<0.5	<0.5
	max	60.5	<0.5	52.0	5.5	<0.5	<0.5
Simulation[a]	min	1.0	<0.5	0.5	0.5	<0.5	<0.5
	med	1.0	<0.5	0.5	0.5	<0.5	<0.5
	max	1.0	<0.5	0.5	0.5	<0.5	<0.5

[a] For MAST, Uppaal and Simulation we have timed the analysis duration by an external tool since the corresponding tools do not support automatic measuring of the analysis time. For these methods a '<0.5' in the table stands for a value below the measuring accuracy of 0.5 seconds.

[b] For Uppaal the analysis times are referred to one single step of binary search.

Tab. 2: Analysis/Verification times in seconds

pitfall for the accuracy of compositional analysis methods. Benchmarks 2 and 4 indicate that holistic analysis approaches are generally more appropriate than modular abstractions in the presence of correlations among task activations and data dependencies. On the other hand, holistic analysis methods are less appropriate for the analysis of timing properties that refer to the actual release time of an event within a large jitter interval. Overall, it is advisable for a system designer to use at least two different formal performance evaluation methods in order not to step in one of the above-mentioned analysis pitfalls.

Moreover, the results show that for the considered benchmarks, the discrete event simulator often provides pretty accurate results, but these results are not necessarily correct, as shown in Benchmark 3 and 5. In other words, by simulation we cannot derive hard bounds for the performance of a system. While for some soft real-time systems this might be tolerable, it is not for systems with hard real-time requirements.

We would also like to emphasize that most of the encountered pitfalls for the analytical techniques are not related to system characteristics that are conceptually impossible to integrate in the respective abstraction. Rather, the analysis difficulties point out aspects that have not yet been investigated for the corresponding methods. In this sense, poor analysis results indicate potential research directions for the improvement of the various formalisms.

Obviously, there are also several questions that the proposed set of small benchmarks cannot answer. For instance, it would be very useful to analyze larger systems with the aim of examining the scalability of the different abstractions with respect to analysis accuracy and analysis times. It could also be interesting to consider the combination of several system properties that have been isolated in the single benchmarks.

2.4 Summary

In this chapter we introduced the reader to the model-based performance evaluation of distributed real-time systems. We identified the deficiencies of empirical performance evaluation methods and stressed the need for formal techniques. We provided an overview of existing formal methods for holistic and compositional performance evaluation in early design stages. Besides analytical approaches, we also indicated state-based methods that can be applied to determine the performance of a system.

In the second part of the chapter we gave a thorough introduction to one particular compositional abstraction, the framework for Modular Performance Analysis (MPA). By means of a simple example, we showed how MPA employs event and resource streams to interface models of individual system components. Based on the general MPA framework, we presented a simple formalism for performance quantification in the time domain. We described its usefulness for the simulation of concrete system executions, but at the same time, we pointed out that it is inappropriate for worst-case performance evaluation. We then described the Real-Time Calculus (RTC), a generalization of the discussed formalism to the interval domain. We discussed the essential elements of RTC such as arrival curves, service curves, and abstract performance components. Finally, we explained how abstract performance models are constructed in RTC, and how they can be used for bounding the performance of distributed systems.

In the last part of the chapter, we provided a quantitative comparison of formal methods for performance evaluation. In particular, we defined a set of benchmark systems, and applied different performance evaluation techniques to these systems. We showed that the results obtained by the various approaches are remarkably different even for apparently basic systems. In other words, we demonstrated that the choice of an appropriate evaluation method matters. But more importantly, we pointed out several pitfalls for the analytical abstractions. These results disclose interesting research problems, such as the proper analysis of cyclic systems (see Chapter 3) or the fight of abstraction losses by means of hybrid analysis approaches (see Chapter 5).

3

Cyclic Dependencies

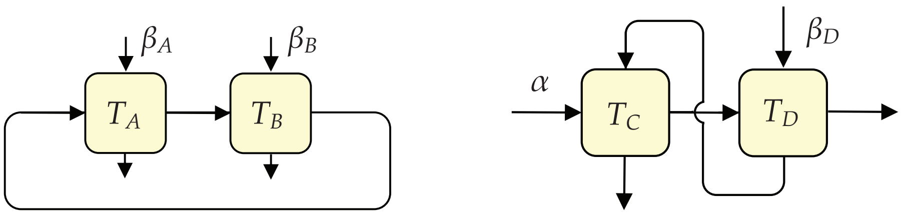
Modular formalisms such as MPA or SymTA/S have shown to be very useful for validating the performance of large and heterogeneous embedded systems. The key for the scalability and low computational complexity of these methods lies in the component-based system representation. In a component-based system model, the interaction among components is abstracted by means of appropriate interfaces. These interfaces enable the incremental validation of very large system architectures by following simple composition mechanisms. However, in the presence of cyclic dependencies among components, the validation of a system is less trivial. A natural way to approach such cyclic dependencies is to apply a fixpoint computation starting from some initial system characterization. However, it is neither known under which conditions such a fixpoint computation converges, nor to what extent the result is faithful to the behaviour of the analyzed system. In this chapter, we develop a general operational semantics underlying the MPA framework, permitting us to answer these questions. We show that the behaviour of systems with non-functional cyclic dependencies can be analyzed by fixpoint iterations. We characterize conditions under which such iterations give safe results, and devise a method that leads to the optimal fixpoint. The results are not limited to MPA, but can be transferred to other modular formalisms.

A large part of the technical contents of this chapter has been originally published in [JPTY08] with important contributions by Prof. Bengt Jonsson.

3.1 Introduction

The formalism of Modular Performance Analysis introduced in Chapter 2 has been successfully used for the analysis of academic and industrial case studies [WTVL06, SPLT10, CLS+06]. All of the considered system models were, however, based on acyclic networks of performance components. Such networks can be analyzed in an incremental fashion by considering each component only once. More specifically, there is at least one component at which the analysis can start, because all inputs of the component are initially known. The computed outputs are then forwarded and utilized for the analysis of dependent system components. This way, an entire system can be progressively analyzed, provided that there are no components which are mutually dependent.

Unfortunately, the analysis of systems with cyclic dependencies is not so straightforward. In this case there exists no component in the component network for which all inputs are initially defined. Hence, it is not obvious how to approach the analysis of the system. Figure 30 shows two examples of systems with cyclic dependencies among components. In general, one can distinguish between functional and non-functional cyclic dependencies. Functional cyclic dependencies are found in systems where the dataflow among components forms cycles. An example of such a system is shown in Figure 30(a). On the other hand, non-functional cyclic dependencies appear if there are opposed dataflow and resource dependencies in a system. An example of a non-functional cyclic dependency is represented in Figure 30(b): component T_C triggers component T_D (dataflow) while component T_D preempts component T_C (resource flow).



(a) Functional cyclic dependency (b) Non-functional cyclic dependency

Fig. 30: Cyclic dependencies in MPA models

An interesting property of cyclic dependencies is that they hinder the abstract performance analysis of a system, but not its simulation. More specifically, the considered physical systems are causal, and hence their simulation is straightforward. On the other hand, the MPA model of a system is purely functional, that is, stateless. Thus, it is not clear how to interpret cycles in an MPA component network.

In various domains, cyclic dependencies can be resolved by means of fixpoint computations. Also in MPA, the most natural way to handle systems with cyclic dependencies is to define some initial event and/or resource streams, and to iterate the analysis until a stable system characterization is reached. For instance, this was the approach followed for the analysis of the benchmark system specified in Section 2.3.3.3. While in practice such fixpoint computations have sometimes resulted in reasonable solutions, its theoretical foundations are completely unclear. For instance, it is not known how to best start a fixpoint iteration in MPA. Also, it is not clear under which conditions the iteration converges, and to what extent the result is trustworthy.

The above questions arise because fundamental issues in linking the abstraction of MPA to an operational semantics have not yet been investigated. In this chapter, we fill this gap by proposing a simple operational model of distributed systems of processes underlying the MPA framework. On this basis, we prove central properties about the faithfulness of fixpoints computed using MPA. The main result is a method that leads to the optimal fixpoint, i.e., to the fixpoint with the smallest abstraction loss. We limit our discussion to non-functional cyclic dependencies. To handle systems with functional cycles in MPA, other abstractions such as dataflow graphs are more appropriate, cf. [TS09].

This chapter is organized as follows. In Section 3.2, the basic problem is detailed by means of a simple example system. In Section 3.3, we present a general operational model of components and streams. We also prove correctness and convergence properties for fixpoint iterations. In Section 3.4, we specialize these results to the MPA framework and discuss techniques for obtaining initial approximations for fixpoint iterations. In Section 3.5, we present the results of a simple experimental evaluation, and comment on the precision of fixpoint computations in MPA. The chapter concludes with an overview on related work in Section 3.6, and a summary in Section 3.7.

3.2 Motivational example

We use the MPA model represented in Figure 31 as a simple motivational example, since it contains a non-functional cyclic dependency which inhibits its compositional analysis. The considered system consists of two tasks, *T1* and *T2*, executed on a processor which implements preemptive fixed priority scheduling. We assume that task *T2* has higher priority than task *T1*. The priorities of the tasks are encoded in the MPA model by appropriately forwarding the stream of processing resources. We assume that the input event traces which trigger *T1* are constrained by the arrival

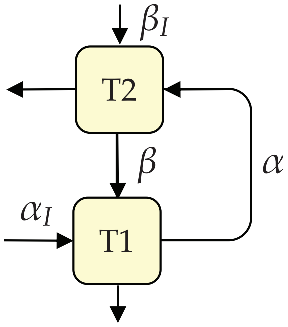


Fig. 31: MPA model of example system

curves $\alpha_I = [\alpha_I^u, \alpha_I^l]$, and that the processing resources of the system are bounded the service curves $\beta_I = [\beta_I^u, \beta_I^l]$. We do not consider any overhead for the context switches between *T1* and *T2*. The goal of the analysis is to determine bounds for the event stream from *T1* to *T2*, and for the resource stream from *T2* to *T1*.

The computation of these streams is not straightforward, as there is a cyclic dependency among the components: *T1* triggers *T2* while *T2* preempts *T1*. We denote a particular characterization of these streams by a pair $\Sigma_S = (\alpha, \beta)$ with $\alpha = [\alpha^u, \alpha^l]$ and $\beta = [\beta^u, \beta^l]$. Similarly, we denote the two input streams from the environment by a pair $\Sigma_I = (\alpha_I, \beta_I)$. Following the dependencies represented in Figure 31, we obtain the equations

$$\alpha = \Phi_\alpha(\alpha_I, \beta) \tag{3.1}$$

$$\beta = \Phi_\beta(\alpha, \beta_I), \tag{3.2}$$

where we use Φ_α and Φ_β to denote the transfer functions of the abstract GPC component, see Equations (2.20)-(2.23). If we ignore the output streams to the environment, the considered system is fully characterized by a pair $\Sigma = (\Sigma_I, \Sigma_S)$. Let us use the letter Ψ to denote the mapping from one characterization Σ to another Σ' by means of the equations (3.1) and (3.2). It is then natural to expect that a characterization of the system behaviour can be obtained as a fixpoint of Ψ, i.e., as a solution to the equation $\Sigma = \Psi(\Sigma)$. A natural way to determine such a fixpoint is to start from some initial approximation Σ^0, and to compute the sequence $\Sigma^0, \Sigma^1, \Sigma^2, \ldots$ with $\Sigma^{k+1} = \Psi(\Sigma^k)$, in the hope that the sequence will converge to a limit Σ^*. However, the correctness of such a fixpoint computation in the context of MPA has not been formally justified so far. In particular, several questions need to be answered:

- Does any fixpoint of Ψ correctly characterize all possible traces of the system?

- Can there be several fixpoints?
- If so, is there an optimal fixpoint (one that provides tighter bounds than all others)?
- Can an (optimal) fixpoint be computed as the limit of a sequence of approximations $\Sigma^0, \Sigma^1, \Sigma^2, ...$?
- Will the iteration always converge to a limit Σ^*?
- How to choose the initial approximation Σ^0?

Let us use the above example to illustrate that fixpoints are in general not unique. Assume that both *T1* and *T2* need one unit of resources to process an event and that the service of the processor corresponds to one resource unit per time unit, i.e., $\beta_I^u(\Delta) = \beta_I^l(\Delta) = \Delta$. Let an input event arrive every second time unit, i.e.,

$$\alpha_I^u(\Delta) = \left\lceil \frac{\Delta}{2} \right\rceil \qquad \text{and} \qquad \alpha_I^l(\Delta) = \left\lfloor \frac{\Delta}{2} \right\rfloor.$$

The optimal fixpoint is the one in which also on the internal event stream α an event arrives every second time unit, i.e., $\alpha = \alpha_I$. It is not difficult to see that this characterization corresponds to the real behaviour of the system. However, there is also a second (much worse) fixpoint for the system which characterizes the internal streams as follows:

$$\alpha^u(\Delta) = \lceil \Delta \rceil, \qquad \alpha^l(\Delta) = 0, \qquad \beta^u(\Delta) = \Delta, \qquad \beta^l(\Delta) = 0.$$

This second fixpoint corresponds to the widest possible bounds for the internal event and resource streams and hence does not contain any information at all.

3.3 Fixpoint Computations on Streams

In order to investigate the remaining questions listed above, in the following we establish a more general framework to specify quantitative properties in component-based systems. This framework can be understood as an abstract description of formalisms such as MPA or SymTA/S.

3.3.1 Operational Model

We consider general systems which consist of components and streams, see Figure 32 for an example.

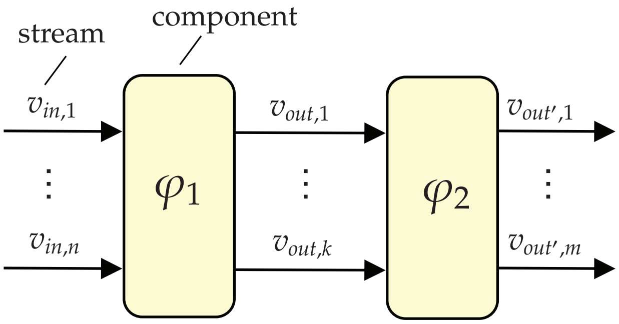


Fig. 32: A general system model

3.3.1.1 Streams

Streams are observable connections between components or between a component and the environment of the system. A stream captures the interaction of a component with its surroundings in terms of data flow or resource flow.

Def. 8: **(Trace)** *A trace on a set V of streams is a function $\sigma : V \mapsto ((\mathbb{R} \times \mathbb{R}) \mapsto \mathbb{R}^{\geq 0})$ which to each stream $v \in V$ assigns a function $\sigma(v)$ from time intervals to observations.*

We take $\mathbb{R}^{\geq 0}$ as the range of observations, since we intend to model the accumulation of some quantity such as events or resource units. For instance, $\sigma(v)(s,t)$ could denote the number of events that have arrived, or the amount of resource units which are available in the time interval $[s,t)$. We assume $\sigma(v)(s,t) + \sigma(v)(t,u) = \sigma(v)(s,u)$ for $s \leq t \leq u$. If there is an initial time point for the evolution of the system, we adapt the above definition by using $(\mathbb{R}^{\geq 0} \times \mathbb{R}^{\geq 0})$ as domain for $\sigma(v)$.[1] We use $Tr(V)$ to denote the set of traces on a set V of streams. The restriction of a trace σ to a subset of streams $V' \subseteq V$ is denoted $\sigma|_{V'}$.

3.3.1.2 Components

In our abstract framework, a component is simply a transducer of traces. It features a set V_I of input streams and a set V_O of output streams.

Def. 9: **(Component trace mapping)** *The behaviour of a component is specified by means of a trace mapping $\varphi : Tr(V_I) \mapsto Tr(V_O)$ which maps any trace σ_I on the input streams V_I to a trace $\varphi(\sigma_I)$ on the output streams V_O.*

---

[1] Without loss of generality, we assume the initial time point to be 0.

3.3.1.3 Systems

A system is a combination of components and streams. It has a set V_I of external input streams, and a set V_O of internal streams and external output streams. All streams of V_O are output streams of some component. They either serve as inputs for other components (internal streams) or go to the system environment (external output streams). In our abstraction, a system is seen as a transducer of traces that combines the behaviour of all its components.

Def. 10: (System trace mapping) *The behaviour of a system is specified by means of a trace mapping* $\psi : Tr(V_I \cup V_O) \mapsto Tr(V_I \cup V_O)$ *which maps any trace on the streams of the system to another trace on system streams. The mapping* ψ *fulfils*

$$\sigma|_{V_I} = \psi(\sigma)|_{V_I} \qquad \forall \sigma \in Tr(V_I \cup V_O), \tag{3.3}$$

that is, it preserves the traces on the input streams. Moreover, for any component of the system with input streams V_{in}*, output streams* V_{out} *and component trace mapping* φ*, the system trace mapping* ψ *observes*

$$\varphi(\sigma|_{V_{in}}) = \psi(\sigma)|_{V_{out}} \qquad \forall \sigma \in Tr(V_I \cup V_O), \tag{3.4}$$

that is, it assigns traces to the internal streams and external output streams according to the trace mappings of the individual components.

We consider only deterministic systems where any trace σ_I on the input streams V_I induces a unique trace σ_O on the output streams V_O. In other words, we suppose a causality relation between inputs and outputs. In the following, we formalize this requirement by defining the property of *simulatability* of a system. Roughly speaking, a system is simulatable if all its components are deterministic, and it does not contain zero-delay cycles. In such a system, for a given input trace, the resulting system trace can be constructed by a stepwise simulation of the system behaviour. For a formal definition of simulatability, we need the following notions:

- A *time vector* on a set of streams V is a function $\tau : V \mapsto \mathbb{R}^{\geq 0}$ that assigns a non-negative time stamp to each stream $v \in V$. We use $\overline{t}$ to denote a time vector which assigns the same time stamp t to all streams $v \in V$.

- For two time vectors τ, τ' on the same set of streams V, $\tau \leq \tau'$ denotes $\tau(v) \leq \tau'(v) \;\; \forall v \in V$.

- Let τ be a time vector on a set of streams $V \subseteq V'$. We say that two traces σ and σ' *agree up to* τ, written as $\sigma \simeq_\tau \sigma'$, if

$$\begin{array}{lcllll} \sigma(v)(s,t) & = & \sigma'(v)(s,t) & \forall v \in V, & \forall s \leq t \leq \tau(v) & \text{and} \\ \sigma(v) & = & \sigma'(v) & \forall v \in V' \setminus V. & & \end{array} \tag{3.5}$$

We use a time vector to express to what extent a trace has been constructed in a simulation. For example, $\tau(v_i)$ denotes the time until which the trace $\sigma(v_i)$ has been determined in the simulation. Note that for two distinct streams v_i and v_j the time stamps $\tau(v_i)$ and $\tau(v_j)$ can be different since in a simulation it may be possible to know the state of a stream further in time with respect to another stream.

Def. 11: **(Simulatable system (bounded past))** *A system is simulatable if for each input trace* $\sigma_I \in Tr(V_I)$ *there is a sequence* $\tau_0, \tau_1, \tau_2, \ldots$ *of time vectors on* V_O *with* $\tau_i \leq \tau_j$ *for* $i < j$, $\tau_0 = \overline{0}$, *and* $\lim_{i\to\infty} \tau_i(v) = \infty \; \forall v \in V_O$, *called the simulation sequence for* σ_I, *such that for all traces* $\sigma, \sigma' \in Tr(V_I \cup V_O)$ *with* $\sigma|_{V_I} = \sigma'|_{V_I} = \sigma_I$ *it holds*

$$\sigma \simeq_{\tau_i} \sigma' \quad \Rightarrow \quad \psi(\sigma) \simeq_{\tau_{i+1}} \psi(\sigma') \quad \forall i \geq 0. \tag{3.6}$$

In essence, a system is simulatable if one can advance time stepwise, and at each step, compute outputs from previously known inputs such that the entire system trace can be determined in ω steps. More specifically, the resulting system trace of a simulatable system is constructed as follows. Let σ_I be a trace on the input streams V_I of the system and let ψ be the trace mapping of the system. Assume that $\tau_0, \tau_1, \tau_2, \ldots$ is the simulation sequence for σ_I. Define a sequence of traces $\sigma^0, \sigma^1, \sigma^2, \ldots$ where σ^0 is any trace such that $\sigma^0|_{V_I} = \sigma_I$ and $\sigma^{i+1} = \psi(\sigma^i)$. Since the system is simulatable, we have $\sigma^i \simeq_{\tau_i} \sigma^{i+j} \;\; \forall i, j \geq 0$. Hence, the sequence $\sigma^0, \sigma^1, \sigma^2, \ldots$ converges to a limit $\sigma = \psi^\omega(\sigma^0)$ which is the resulting system trace for the input σ_I.

3.3.1.4 Extensions for unbounded past

Let us now extend the above concepts to the case where the past is unbounded, that is, where there is no initial time point for the evolution of a system. This case can be handled by adapting the above definitions as follows:

- A time vector is a function $\tau : V \mapsto \mathbb{R}$ that assigns a time stamp to each stream $v \in V$.
- For a given time vector τ, we say that σ^0 is a *possible system trace up to* τ if $\sigma^0 \simeq_\tau \psi(\sigma^0)$. We denote the set of traces σ such that $\sigma \simeq_\tau \sigma^0$ as $Cont(\sigma^0, \tau)$.

Note that now a time vector can also assign negative time stamps to streams. In other words, time stamps can reach arbitrarily far into the past. The set $Cont(\sigma^0, \tau)$ can be interpreted as the set of all possible continuations of the trace σ^0 after the time vector τ.

Next, we define the concept of simulatability from a given time vector.

Def. 12: (Simulatable system from time vector) *Consider a system with a set of streams $V = V_I \cup V_O$. Let τ be an arbitrary real-valued time vector on V_O. The system is simulatable from τ if, whenever σ^0 is a possible system trace up to τ, there is a sequence $\tau_0, \tau_1, \tau_2, \ldots$ of time vectors on V_O with $\tau_i \leq \tau_j$ for $i < j$, $\tau_0 = \tau$, and $\lim_{i\to\infty} \tau_i(v) = \infty\ \forall v \in V_O$, called the simulation sequence for σ^0 from τ, such that for all traces $\sigma, \sigma' \in Tr(V_I \cup V_O)$ with $\sigma|_{V_I} = \sigma'|_{V_I} = \sigma_I$ it holds*

$$\sigma \simeq_{\tau_i} \sigma' \quad \Rightarrow \quad \psi(\sigma) \simeq_{\tau_{i+1}} \psi(\sigma') \quad \forall i \geq 0. \tag{3.7}$$

The difference with respect to Definition 11 is that the simulation sequence does not start any longer at the origin of the time axis, but at a point in time until which possible system traces are defined.

Def. 13: (Simulatable system (unbounded past)) *A system is simulatable if it is simulatable from any real-valued time vector τ.*

In essence, a system with unbounded past is simulatable if it is always possible to extend a partially defined system trace by means of simulation. For a system that is simulatable from τ, we can construct the resulting system trace $\sigma = \psi^{\omega}(\sigma^0)$ in the same way as above, starting from σ^0 which is now a possible system trace up to τ, i.e., $\sigma \simeq_{\tau} \sigma^0$.

3.3.1.5 Abstract characterization of streams, components and systems

MPA or SymTA/S do not reason about single traces of a system. Rather, they use sets of traces to characterize streams. As we have seen in Section 2.2.3, this is the main prerequisite for worst-case performance evaluation. In this section, we extend our abstract operational model to this kind of stream characterization.

Def. 14: (Stream characterization) *A characterization on a set V of streams is a function $\Sigma : V \mapsto 2^{((\mathbb{R}\times\mathbb{R})\mapsto\mathbb{R}^{\geq 0})}$ which assigns a set of traces $\Sigma(v)$ to each stream $v \in V$.*

For instance, a characterization Σ could specify upper and lower bounds on the number of events that can arrive in a given time interval. Similarly, it could specify bounds for the amount of resources which are available in a given time interval. We use $Char(V)$ to denote the set of characterizations on a set V of streams. The restriction of a characterization Σ to a subset of streams $V' \subseteq V$ is denoted $\Sigma|_{V'}$.

Def. 15: (Conformance of a trace w.r.t. a stream characterization) *Let σ be a trace on a set V of streams. Let Σ be a characterization on the same set V of streams. Then, we say that σ conforms to Σ, denoted as $\sigma \models \Sigma$, if $\sigma(v) \in \Sigma(v)\ \ \forall v \in V$.*

If a trace σ conforms to a characterization Σ, we also say that σ *satisfies* Σ.

Def. 16: **(Satisfiable characterization)** *Let Σ be a characterization on a set V of streams. Σ is satisfiable if there exists a trace σ on V such that $\sigma \models \Sigma$.*

The action of a component can now be seen as a transformation of stream characterizations.

Def. 17: **(Component characterization mapping)** *Consider a component with input streams V_I, output streams V_O, and trace mapping φ. The component can be specified by a characterization mapping $\Phi : Char(V_I) \mapsto Char(V_O)$ from characterizations on the set V_I of input streams to characterizations on the set V_O of output streams. The mapping Φ is correct with respect to φ, i.e., it has the property*

$$\sigma_I(v_i) \models \Sigma_I(v_i) \ \ \forall v_i \in V_I \quad \Rightarrow \quad \varphi(\sigma_I)(v_o) \models \Phi(\Sigma_I)(v_o) \ \ \forall v_o \in V_O. \tag{3.8}$$

Similarly as for trace mappings, we can combine all component characterization mappings to form a system characterization mapping.

Def. 18: **(System characterization mapping)** *Consider a system with streams $V_I \cup V_O$ and system trace mapping ψ. The system can be specified by means of a characterization mapping $\Psi : Char(V_I \cup V_O) \mapsto Char(V_I \cup V_O)$ which maps any characterization of the system streams to another characterization of the system streams. The mapping Ψ is correct with respect to ψ, that is, it fulfils*

$$\sigma \models \Sigma \quad \Rightarrow \quad \psi(\sigma) \models \Psi(\Sigma) \quad \forall \sigma \in Tr(V_I \cup V_O), \Sigma \in Char(V_I \cup V_O). \tag{3.9}$$

3.3.2 Correctness and Convergence

In this section we use the above general operational model to ascertain the iterative computation of system characterizations. More specifically, we investigate the correctness and convergence of such iterations. We first discuss systems with an explicit starting point in time, and thereafter extend the results to the case of unbounded past.

3.3.2.1 Correctness

Consider a simulatable system with a set of input streams V_I and set of internal streams and output streams V_O. Let ψ be the trace mapping of the system and let Ψ be a characterization mapping which is correct with respect to ψ. Assume that Σ_I is a characterization on V_I that bounds the input traces of the system. Let σ_I be any input trace with $\sigma_I \models \Sigma_I$. The basic question we need to study is whether the trace $\psi^\omega(\sigma_I)$, i.e., the resulting system trace on input σ_I, conforms to the limit of a sequence $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), ...$ In other words, we want to know whether the iteration on system characterizations is correct with respect to the resulting system trace $\psi^\omega(\sigma_I)$.

In the following, we answer this question by relating the iteration on system characterizations to the evolution of system traces. To do so, it is helpful to link the definition of trace conformance to a time vector.

Def. 19: **(Conformance of trace up to time vector)** *Let τ be a time vector and Σ a characterization on V. A trace σ conforms to Σ up to τ, denoted as $\sigma \models_{\leq\tau} \Sigma$ if there exists a trace σ' such that $\sigma' \models \Sigma$ and $\sigma \simeq_\tau \sigma'$.*

We can now show that the iteration on an appropriate initial system characterization is correct with respect to the behaviour of the system.

Thm. 2: *Let $\tau_0, \tau_1, \tau_2, \ldots$ be the simulation sequence for σ_I and let $\psi^\omega(\sigma_I)$ be the resulting system trace. If the characterization Σ^0 is satisfiable and $\Sigma^0|_{V_I} = \Sigma_I$ then*

$$\psi^\omega(\sigma_I) \models_{\leq\tau_i} \Psi^i(\Sigma^0) \qquad \forall i \geq 0. \tag{3.10}$$

Proof. The system trace $\psi^\omega(\sigma_I)$ is the limit of a sequence $\sigma^0, \sigma^1, \sigma^2, \ldots$ with $\sigma^0|_{V_I} = \sigma_I$ and $\sigma^{i+1} = \psi(\sigma^i)$. We can prove by induction that $\sigma^i \models_{\leq\tau_i} \Psi^i(\Sigma^0)$. The base case $\sigma^0 \models_{\leq\bar{0}} \Sigma^0$ results from $\sigma^0|_{V_I} = \sigma_I$, $\Sigma^0|_{V_I} = \Sigma_I$, and the satisfiability of Σ^0. For the inductive step, assume that $\sigma^i \models_{\leq\tau_i} \Psi^i(\Sigma^0)$, i.e., there is a trace σ' with $\sigma^i \simeq_{\tau_i} \sigma'$ such that $\sigma' \models \Psi^i(\Sigma^0)$. Since ψ is correct with respect to Ψ, we obtain $\psi(\sigma') \models \Psi^{i+1}(\Sigma^0)$. Moreover, $\psi(\sigma^i) \simeq_{\tau_{i+1}} \psi(\sigma')$ because the system is simulatable. Hence, we get $\psi(\sigma^i) \models_{\leq\tau_{i+1}} \Psi^{i+1}(\Sigma^0)$. The conclusion of the theorem follows by noting that $\psi^\omega(\sigma_I) \simeq_{\tau_i} \sigma^i$ $\forall i \geq 0$.

□

Intuitively, Theorem 2 states that the behaviour of a system can be safely bounded by iteratively applying the system characterization mapping Ψ to a satisfiable initial system characterization Σ^0 which is compatible with the specification of the input streams Σ_I. In other words, if the iteration $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), \ldots$ converges, then the fixpoint of the iteration is a correct characterization of the system behaviour.

3.3.2.2 Convergence

Theorem 2 does not say anything about convergence of the iteration. For practical reasons, we are obviously also interested in knowing when the sequence $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), \ldots$ converges and, in particular, in how to choose Σ^0. To investigate these questions, let us ascertain the ordering of stream characterizations.

Def. 20: **(Partial order on *Char*(V))** *Let V be a set of streams. The conformance relation $\models$ establishes a natural partial order $\sqsubseteq$ on the set of all characterizations $Char(V)$ which is defined as*

$$\Sigma \sqsubseteq \Sigma' \quad \Leftrightarrow \quad (\ \sigma \models \Sigma \ \Rightarrow \ \sigma \models \Sigma' \quad \forall \sigma \in Tr(V)\ ) \tag{3.11}$$

We assume that the considered systems fulfil the following conditions:

1. The system is simulatable.
2. The set $Char(V)$ with the partial order $\sqsubseteq$ forms a chain-complete partially ordered set, that is, any chain $\Sigma^0 \sqsubseteq \Sigma^1 \sqsubseteq \Sigma^2 \sqsubseteq ...$ has a least upper bound $\bigsqcup_{i \geq 0} \Sigma^i$.
3. The satisfaction of a characterization Σ is a safety property, that is,
$$\sigma \models_{\leq \tau} \Sigma \ \ \forall \tau \quad \Rightarrow \quad \sigma \models \Sigma. \tag{3.12}$$
4. For any trace $\sigma \in Tr(V)$ there is a least (strongest) characterization Σ_σ such that $\sigma \models \Sigma_\sigma$.
5. The system characterization mapping Ψ is monotone, that is,
$$\Sigma \sqsubseteq \Sigma' \quad \Rightarrow \quad \Psi(\Sigma) \sqsubseteq \Psi(\Sigma'). \tag{3.13}$$
6. The system characterization mapping Ψ is continuous, that is,
$$\Psi(\bigsqcup_{i \geq 0} \Sigma^i) = \bigsqcup_{i \geq 0} \Psi(\Sigma^i) \tag{3.14}$$
for any chain $\Sigma^0 \sqsubseteq \Sigma^1 \sqsubseteq \Sigma^2 \sqsubseteq ...$.

Based on the above assumptions, we can draw formal conclusions on the convergence of iterative applications of Ψ, as well as on the optimality of fixpoints.

Thm. 3: *Among all characterizations Σ such that $\Sigma|_{V_I} = \Sigma_I$ and that are satisfiable by at least one actual system trace, Ψ has a unique smallest fixpoint Σ^* which is satisfied by all traces σ with $\sigma|_{V_I} \models \Sigma_I$.*

Proof. Let σ be some actual system trace with $\sigma|_{V_I} \models \Sigma_I$, and let Σ_σ be the strongest characterization of σ with $\Sigma_\sigma|_{V_I} = \Sigma_I$. Since Ψ is correct with respect to ψ, we have $\psi(\sigma) = \sigma \models \Psi(\Sigma_\sigma)$ and hence $\Sigma_\sigma \sqsubseteq \Psi(\Sigma_\sigma)$. By monotonicity of Ψ, we obtain $\Psi^k(\Sigma_\sigma) \sqsubseteq \Psi^{k+1}(\Sigma_\sigma) \ \forall k \geq 0$. This means that the sequence $\Sigma_\sigma, \Psi(\Sigma_\sigma), \Psi^2(\Sigma_\sigma)...$ converges to a fixpoint Σ^* which is the least upper bound of the chain $\Sigma_\sigma \sqsubseteq \Psi(\Sigma_\sigma) \sqsubseteq \Psi^2(\Sigma_\sigma) \sqsubseteq ...$. Moreover, because of Theorem 2 and the assumption that the satisfaction of a characterization is a safety property, it holds that $\sigma' \models \Sigma^*$ for any system trace σ' with $\sigma'|_{V_I} \models \Sigma_I$. It remains to show that the same fixpoint is reached no matter which system trace is used for constructing the initial characterization. Consider two distinct system traces σ_1, σ_2 with $\sigma_1|_{V_I} \models \Sigma_I$ and $\sigma_2|_{V_I} \models \Sigma_I$. Let $\Sigma_{\sigma_1}, \Sigma_{\sigma_2}$ be the corresponding strongest characterizations. Let Σ^*_1, Σ^*_2 denote the fixpoints reached by the iterations $\Sigma_{\sigma_1}, \Psi(\Sigma_{\sigma_1}), \Psi^2(\Sigma_{\sigma_1}), ...$ and

$\Sigma_{\sigma_2}, \Psi(\Sigma_{\sigma_2}), \Psi^2(\Sigma_{\sigma_2}), ...$, respectively. Due to Theorem 2 we have that Σ_1^* covers all possible system traces. In particular, it must hold $\Sigma_{\sigma_2} \sqsubseteq \Sigma_1^*$. By monotonicity of Ψ, we obtain $\Psi^k(\Sigma_{\sigma_2}) \sqsubseteq \Psi^k(\Sigma_1^*) = \Sigma_1^* \ \forall k \geq 0$ and hence $\Sigma_2^* \sqsubseteq \Sigma_1^*$. By applying the same reasoning to Σ_{σ_1} and Σ_2^*, we get $\Sigma_1^* \sqsubseteq \Sigma_2^*$ and hence $\Sigma_1^* = \Sigma_2^*$.

□

The above theorem guarantees convergence to the optimal fixpoint if the particular characterization Σ_σ is used as starting point for the iteration where σ is some actual system trace. It is, however, also possible to generalize this result to initial characterizations Σ^0 with $\Sigma_\sigma \sqsubseteq \Sigma^0 \sqsubseteq \Sigma^*$.

Cor. 1: *The optimal fixpoint Σ^* of Ψ can be obtained as the limit of a sequence $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), ...$ of characterizations, provided that*

- $\Sigma^0|_{V_I} = \Sigma_I$
- Σ^0 *is satisfied by at least one actual system trace*
- $\Sigma^0 \sqsubseteq \Sigma^*$

Proof. By assumption we have $\Sigma_\sigma \sqsubseteq \Sigma^0 \sqsubseteq \Sigma^*$. Since Ψ is monotone, we obtain $\Psi^k(\Sigma_\sigma) \sqsubseteq \Psi^k(\Sigma^0) \sqsubseteq \Sigma^* \ \forall k \geq 0$. It follows that also the sequence $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0)...$ converges to Σ^*.

□

Intuitively, Theorem 3 and Corollary 1 state that there is a unique least fixpoint among all characterizations of a system which are compatible with the specification of the input streams. Further, this fixpoint can be obtained by iterating on a system characterization which is satisfied by an actual system trace. Hence, if we can construct one actual system trace, then we can also bound all possible traces of the system by means of a fixpoint computation.

3.3.2.3 Extensions for unbounded past

The above results on correctness and convergence of fixpoint iterations can be easily extended to systems without an explicit starting point, that is, with unbounded past. The basic idea for adapting Theorems 2, 3 and Corollary 1 to this case is to consider possible system traces up to a general time vector τ.

Thm. 4: *Let σ^0 be a possible system trace up to a time vector τ with $\sigma^0|_{V_I} \models \Sigma_I$. Let $\tau_0, \tau_1, \tau_2, ...$ be the simulation sequence for σ^0 from τ. If $\sigma^0 \models \Sigma^0$, then*

$$\psi^\omega(\sigma^0) \models_{\leq \tau_i} \Psi^i(\Sigma^0) \qquad \forall i \geq 0. \tag{3.15}$$

Proof. Analogous to the proof of Theorem 2.

□

Thm. 5: *Let σ^0 be a possible system trace up to a time vector τ with $\sigma^0|_{V_I} \models \Sigma_I$. Among all characterizations Σ such that $\Sigma|_{V_I} = \Sigma_I$ and that are satisfied by at least one actual system trace in $Cont(\sigma^0, \tau)$, Ψ has a unique smallest fixpoint Σ^* which is satisfied by all traces $\sigma \in Cont(\sigma^0, \tau)$ with $\sigma|_{V_I} \models \Sigma_I$.*

Proof. Analogous to the proof of Theorem 3.

□

Cor. 2: *The optimal fixpoint Σ^* of Ψ can be obtained as the limit of a sequence $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), \ldots$ of characterizations, provided that*

- $\Sigma^0|_{V_I} = \Sigma_I$
- Σ^0 *is satisfied by at least one actual system trace in* $Cont(\sigma^0, \tau)$
- $\Sigma^0 \sqsubseteq \Sigma^*$

Proof. Analogous to the proof of Corollary 1.

□

3.4 Fixpoint Computations in MPA

Let us now discuss how to transfer the results of Section 3.3 to the formalism of MPA with RTC. The aim is to guarantee the correctness and convergence of fixpoint iterations in the framework of MPA.

3.4.1 Verification of Assumptions

Consider a general MPA system S with a set of streams $V = V_r \cup V_c$ where V_r represents the set of event streams and V_c the set of resource streams. In the MPA view, a trace σ on V corresponds to the assignment of an arrival function $r(s,t)$ to each event stream $v \in V_r$ and of a service function $c(s,t)$ to each resource stream $v \in V_c$. Similarly, a characterization on V corresponds to the assignment of a tuple of arrival curves $\alpha = [\alpha^u, \alpha^l]$ to each event stream $v \in V_r$ and of a tuple of service curves $\beta = [\beta^u, \beta^l]$ to each resource stream $v \in V_c$. In MPA, the trace mapping φ and the characterization mapping Φ of a system component are specified by appropriate transfer functions $(r', c') = \varphi(r, c)$ and $(\alpha', \beta') = \Phi(\alpha, \beta)$, see Sections 2.2.2.2 and 2.2.3.4. The trace mapping ψ and characterization mapping Ψ of an MPA system are then simply obtained by combining all transfer functions φ and Φ of the individual MPA components.

In order to carry over the results on correctness and convergence of fixpoint iterations to MPA, we need to carefully verify that all assumptions made in Section 3.3 are met by MPA systems. In particular, we must ensure that the following requirements are met:

1. The MPA system S is simulatable.

   In general, not every system is simulatable. We can, however, define two practically unrestrictive conditions which are sufficient to guarantee simulatability:

   - For each MPA component of S, the time needed to process an input event is > 0.
   - S does not have any cycle of resource streams.

   The above conditions can be informally justified as follows. For each component, the future output can be predicted until the next point in time when it receives an input event, or until the availability of resources changes. Hence, given the input traces of S and an arbitrary time point t, it is possible to infer a simulation sequence for S uniquely, that is, an increasing sequence of time points $t_0 = t, t_1, t_2, ...$ at which some event or resource streams of the system evolve.

2. The tuple $(Char(V), \sqsubseteq)$ forms a chain-complete poset.

   This condition requires that every chain of characterizations $\Sigma^0 \sqsubseteq \Sigma^1 \sqsubseteq \Sigma^2 \sqsubseteq ...$ has a least upper bound. Translated to the domain of MPA, it means that the sets of upper arrival curves and upper service curves need to have upper bounds and, analogously, the sets of lower arrival curves and lower service curves need to have lower bounds. The sets of lower arrival curves and lower service curves have the implicit lower bounds $\alpha^l(\Delta) = 0 \; \forall \Delta > 0$ and $\beta^l(\Delta) = 0 \; \forall \Delta > 0$, respectively. However, the set of upper arrival curves has no natural upper bound. The reason is that for any given upper arrival curve α_1^u it is always possible to find another curve α_2^u such that $\alpha_1^u(\Delta) > \alpha_2^u(\Delta) \;\; \forall \Delta > 0$. Therefore, we augment the set of upper arrival curves by all curves α^u with $\alpha^u(\Delta) = \infty \; \forall \Delta \geq \Delta_0$ for some $\Delta_0 > 0$. This extension is not necessary for the set of upper service curves, since any resource availability is upper bounded by a curve $\beta^u(\Delta) = c \cdot \Delta$ where c represents the maximum bandwidth of the resource, i.e., the maximum amount of service units provided per time unit. As a result, we can guarantee that the extended set of stream characterizations in MPA forms a chain-complete poset.

3. The satisfaction of a characterization Σ is a safety property.

   In the setting of MPA, the requirement (3.12) translates to

$$\begin{array}{lll} \alpha^l(t-s) \leq r(s,t) \leq \alpha^u(t-s) \;\; \forall s \leq t & \Rightarrow & r \models \alpha \\ \beta^l(t-s) \leq c(s,t) \leq \beta^u(t-s) \;\; \forall s \leq t & \Rightarrow & c \models \beta, \end{array} \tag{3.16}$$

   which is satisfied by definition of arrival and service curves (cf. Relations (2.9) and (2.15)).

4. For any trace σ there is a strongest characterization Σ_σ with $\sigma \models \Sigma_\sigma$.

   In MPA, the strongest characterization of a trace is obtained by means of Equations (2.11), (2.12), (2.17), and (2.18).

5. The system characterization mapping Ψ is monotone.

   For the sake of simplicity, we limit the verification of this requirement to a single GPC component as described in Section 2.2.3.4. Let us abbreviate the transfer functions (2.20)-(2.23) with

$$\alpha' = \Phi_{\mathrm{GPC},\alpha}(\alpha,\beta) \qquad \text{and} \qquad \beta' = \Phi_{\mathrm{GPC},\beta}(\alpha,\beta).^2$$

   We have to verify that

$$\begin{array}{rcll} R^{\alpha_1} \subseteq R^{\alpha_2} & \Rightarrow & R^{\Phi_{\mathrm{GPC},\alpha}(\alpha_1,\beta)} \subseteq R^{\Phi_{\mathrm{GPC},\alpha}(\alpha_2,\beta)} & \forall \alpha_1,\alpha_2,\beta \\ C^{\beta_1} \subseteq C^{\beta_2} & \Rightarrow & R^{\Phi_{\mathrm{GPC},\alpha}(\alpha,\beta_1)} \subseteq R^{\Phi_{\mathrm{GPC},\alpha}(\alpha,\beta_2)} & \forall \alpha,\beta_1,\beta_2 \\ R^{\alpha_1} \subseteq R^{\alpha_2} & \Rightarrow & C^{\Phi_{\mathrm{GPC},\beta}(\alpha_1,\beta)} \subseteq C^{\Phi_{\mathrm{GPC},\beta}(\alpha_2,\beta)} & \forall \alpha_1,\alpha_2,\beta \\ C^{\beta_1} \subseteq C^{\beta_2} & \Rightarrow & C^{\Phi_{\mathrm{GPC},\beta}(\alpha,\beta_1)} \subseteq C^{\Phi_{\mathrm{GPC},\beta}(\alpha,\beta_2)} & \forall \alpha,\beta_1,\beta_2, \end{array}$$

   where R^α denotes the set of all event traces that conform to α, and C^β the set of all resource patterns that conform to β. These propositions are clearly satisfied as for a GPC component an extension of the input traces cannot lead to a restriction of the output traces.

6. The system characterization mapping Ψ is continuous.

   By applying similar reasoning as for monotonicity, also this requirement can be ensured for MPA components.

The above properties ensure that Theorems 2, 3, 4, 5, as well as Corollaries 1 and 2 can be directly applied to guarantee the correctness and convergence of fixpoint iterations in MPA. There is, however, one more thing to take into account: As stated by the theorems of Section 3.3, the iteration converges to the optimal fixpoint Σ^* only if the initial characterization Σ^0 of the system is satisfied by an actual system trace and $\Sigma^0 \sqsubseteq \Sigma^*$. How to get such a characterization will be discussed next.

---

[2] In the case of bounded past we consider Equations (2.28) and (2.29) instead of Equations (2.21) and (2.22).

3.4.2 Obtaining an Initial Characterization

The simplest way to obtain Σ^0 is to construct an actual system trace σ by means of a simulation and then to determine its strongest specification Σ_σ. The simulation of an MPA model can be performed with various discrete event simulators. For instance, system-level simulators such as PESIMDES [Per] (limited to PJD inputs) or the Real-Time Simulation Toolbox [TS] can be used to this end. The task is simplified by the fact that any arbitrary system trace is adequate, as long as it conforms to the specification of the input streams Σ_I. Hence, input traces with highly regular patterns, e.g. periodic input traces, can often be employed to trigger the simulation.

An alternative way for constructing an initial characterization Σ^0 is to employ analytic techniques such as the method proposed by Schiøler et al. in [SJDL05]. More specifically, in [JPTY08] we further strengthen the theorems of Section 3.3 such that the convergence of a fixpoint iteration can be ensured whenever the initial characterization Σ^0 is satisfiable, that is, even if Σ^0 is not satisfied by an actual system trace. This extension permits us to start fixpoint iterations with a characterization Σ^0, derived from long-term rates of system streams, as first proposed in [SJDL05].

3.4.3 Summary of the Method

Let us summarize the proposed methodology for analyzing MPA systems with non-functional cyclic dependencies.

1. Construct some trace σ for the system such that $\sigma|_{V_I} \models \Sigma_I$ and $\sigma \models \Sigma^*$. Such a trace can be determined by means of a simulation. The task is made easier by the fact that only one trace is needed and that the trace can be chosen as regularly as possible.

2. Determine the strongest characterization Σ_σ for σ by means of Equations (2.11), (2.12), (2.17), and (2.18). Use Σ_σ as initial characterization Σ^0.

3. Perform the fixpoint iteration $\Sigma^0, \Psi(\Sigma^0), \Psi^2(\Sigma^0), ...$, that is, apply the transfer functions of all MPA components until a stable characterization Σ^* is reached. Σ^* is the optimal fixpoint of Ψ.

An alternative to steps 1. and 2. is to construct an initial characterization Σ^0 for the system by the analytic technique of [SJDL05].

In short, the recommended way to compute fixpoints is to start from a strong initial characterization which is included in the sought optimal fixpoint and thereafter to iterate towards a fixpoint. We observe that the dual approach - starting from a weak initial characterization and iterating towards a stronger solution - will often lead to poor precision, as illustrated by the example in Section 3.2.

3.5 Experiments

In this section, we show how the fixpoint iteration described above succeeds in the analysis of a concrete example system with a non-functional cyclic dependency. Consider the system depicted in Figure 33. It consists of a sequence of three tasks *T1*, *T2* and *T3* that process a periodic event stream α_I . The system has two processing resources with resource availability $\beta_I(\Delta) = \beta_{II}(\Delta) = \Delta$. The first processor is shared by *T1* and *T3* and implements preemptive fixed priority scheduling with *T3* having higher priority than *T1*. The second processor executes *T2* only. The system contains a non-functional cyclic dependency since *T1* indirectly triggers *T3* while *T3* preempts *T1*.

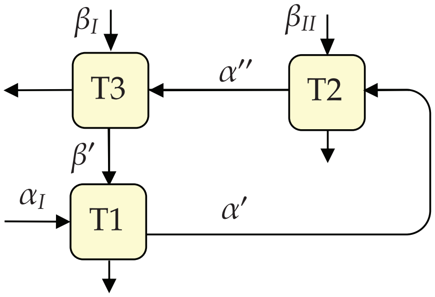


Fig. 33: MPA model for experiment

We assume that the input event stream α_I is strictly periodic with a period P of 10 time units, that is,

$$\alpha_I^u(\Delta) = \left\lceil \frac{\Delta}{10} \right\rceil \qquad \text{and} \qquad \alpha_I^l(\Delta) = \left\lfloor \frac{\Delta}{10} \right\rfloor.$$

Further, we assume that *T1*, *T2* and *T3* have constant event processing times of 4, 7 and 5 time units, respectively. We want to compute bounds for the event streams α', α'' and for the service stream β' shown in Figure 33.

In order to find an appropriate initial characterization for the fixpoint iteration, we first simulate the system execution. For this simulation we use the PESIMDES tool [Per]. Since the system architecture is simple, the simulation could also be performed by hand without much effort. We observe that after an initial transitory phase, task *T1* produces output events for task *T2* with a recurring timing pattern. In particular, it produces events with consecutive distances of 4 time units, 16 time units, 4 time units, 16 time units etc. This makes it easy to characterize the trace σ of the event stream *T1-T2*, and find the strongest characterization Σ_σ of this trace in terms of upper and lower arrival curves. The corresponding arrival curves are depicted in Figure 34.

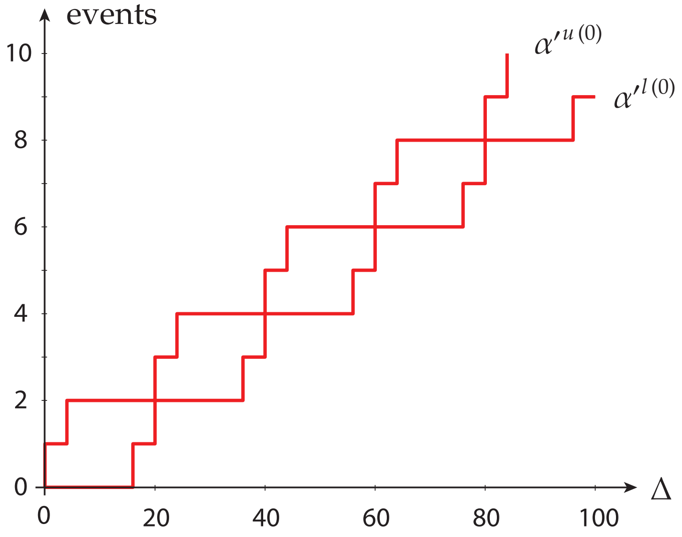


Fig. 34: Strongest characterization of the simulated trace *T1-T2*

We then use these arrival curves as initial characterization for the fixpoint iteration in MPA. More specifically, we model the tasks *T1*, *T2* and *T3* with three GPC components, and repeatedly compute their outgoing arrival and service curves according to Equations (2.20)-(2.23), and in the order *T2*, *T3*, *T1*. After 4 iterations we reach a fixpoint, that is, all the arrival curves and service curves in the MPA model are stable. The iteration sequence for α', α'', and β' as well as the final characterizations of these streams are shown in Figures 35, 36(a), and 36(b), respectively.

Note that the optimal fixpoint for the system characterization mapping Ψ does not necessarily correspond to a tight characterization of the individual streams in a system. For example, in the considered system, the initial characterization $\alpha'^{(0)}$ is satisfied not only by the simulated system trace σ but by all traces that the system can exhibit. In fact, these traces differ only by the arrival time of the first event, as this is the only non-deterministic element of the system. In other words, $\alpha'^{(0)}$ is a tight characterization of the stream *T1-T2*. Nevertheless, in Figure 35 we can observe that the performed fixpoint iteration leads to a considerably weaker, i.e. less accurate, characterization $\alpha'^{(*)}$. Similar reasonings apply to the streams α'' and β'.

The potentially poor precision of fixpoint solutions in compositional performance analysis was already pointed out in Section 2.3.4.3. The

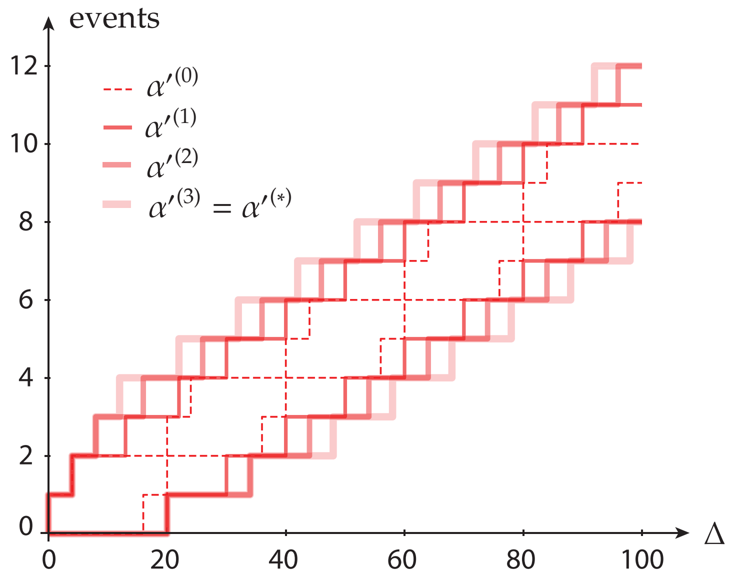
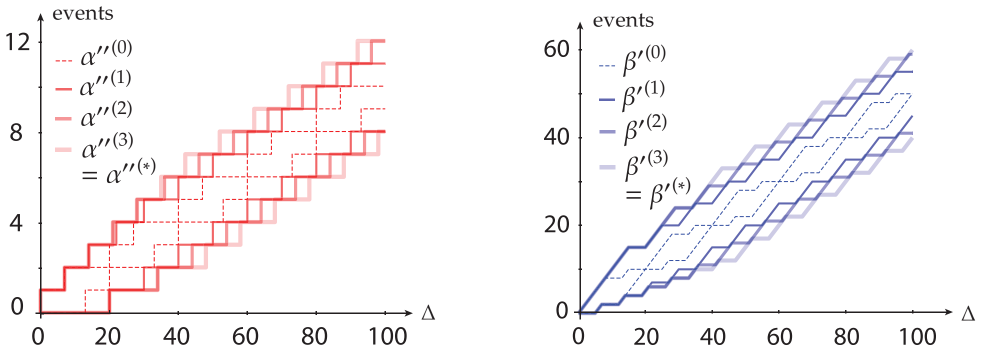


Fig. 35: Iteration sequence for the stream α'



(a) (b)

Fig. 36: Iteration sequences for the streams α'' (a) and β' (b)

reasons for this common effect are not yet fully clarified. For sure the implicit abstraction losses of the transfer functions (2.20)-(2.21) play an important role. Nevertheless, it is not clear why in fixpoint solutions the abstraction losses are often particularly pronounced.

Moreover, the accuracy of fixpoint characterizations basically seems to be unpredictable. In order to illustrate this, let us slightly modify the above example system by reducing the processing time of task $T2$ to 2 time units. By following the same analysis approach as before, we obtain the iteration sequence of Figure 37 for α', where we stopped the fixpoint computation after 30 iterations. In the figure, successive iterations are represented in alternate colors. The figure shows that with the proceeding iteration, the characterization of α' becomes progressively weaker and finally looses all information about the stream. In essence, the experiment shows that small variations in the configuration of a cyclic system may have a strong impact on the results of the performance analysis.

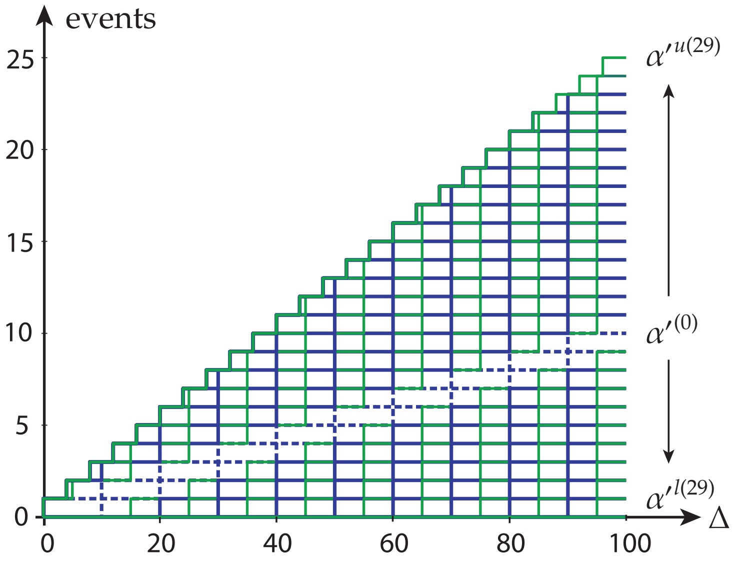


Fig. 37: Iteration sequence for the characterization α' in the modified system

3.6 Related Work

The general topic of modelling (cyclic) systems as transformers of streams goes back to Kahn process networks [Kah74] (extension to real-time in [Yat93]). The major difference between such dataflow models and our operational model is that we do not compute fixpoints to determine the actual behaviour of a system. Rather, we consider fixpoints of *constraints* on system behaviours. Consequently, our model can be employed as operational semantics for quantitative worst-case formalisms such as MPA or SymTA/S.

Within the domain of compositional performance evaluation, Jersak et al. [JRE05] first proposed using fixpoint iterations to handle cyclic systems in the context of SymTA/S. They consider the special case of systems in which event streams are abstracted by means of PJD event models. The approach is limited in terms of the underlying abstraction in the sense that it cannot be extended to other compositional formalisms such as MPA. It makes only informal statements about convergence properties of fixpoint iterations. Moreover, it does not discuss the influence of initial system approximations on the result of fixpoint iterations.

Schiøler et al. presented a method for the analysis of cyclic systems with Network Calculus [SJDL05]. The method employs analytical long-term rates of streams to construct an initial characterization of a system, and thereafter iterates towards a fixpoint. The authors of the method show that, under some assumptions, the iteration converges to an optimal fixpoint. However, the method does not explicitly define an operational model of system behaviours. In addition, the statement on correctness of fixpoints seems to rely on unstated assumptions regarding causality or absence of zero-delay cycles.

Following the original publication of our results in [JPTY08], Stein et al. established a similar mathematical framework to prove the correctness and convergence of fixpoint iterations in the SymTA/S approach [SDI$^+$08]. Under similar assumptions, e.g. monotonicity and continuity of transfer functions, they prove that the SymTA/S analysis converges to an optimal fixpoint. Furthermore, they show that in an iteration, the same fixpoint is found irrespective of the particular sequence of local component evaluations.

Lastly, in [TS09] Thiele et al. described a method to handle functional cycles in MPA. The method extends the MPA analysis framework to cyclic dataflow graphs such as SDF (synchronous dataflow), or marked graphs. The approach enables the analysis of applications that are specified as dataflow graphs, and executed on distributed systems with resource sharing mechanisms such as Fixed Priority (FP) or Time Division Multiple Access (TDMA). As the method is very general, it permits the designer to analyze many practically relevant scenarios such as systems with finite buffers and back-pressure effects due to blocking write semantics.

3.7 Summary

In this chapter we discussed the performance analysis of distributed systems with non-functional cyclic dependencies. We formally justified the natural approach of using fixpoint iterations to handle such dependencies in compositional performance analysis. More specifically, we proved that fixpoint iterations in MPA are correct in the sense of faithful to the behaviour of the system, if the initial system characterization is chosen appropriately, and the system meets some mild assumptions such as being simulatable. We also devised conditions under which fixpoint iterations in MPA converge. The main result is a method that leads to the optimal fixpoint of a system. Our findings are not limited to MPA but can be transferred to other compositional formalisms such as SymTA/S.

In order to obtain these results, we developed a general operational model underlying the MPA framework. The model represents stream-oriented distributed systems in an abstract and compositional fashion. We first looked at systems with an explicit starting point on the time axis and then generalized the results to the case of unbounded past. Our research indicates that the recommended way to determine fixpoints is to start from a strong initial characterization of the system, which is included in the sought optimal fixpoint, and then to iterate towards the fixpoint while obtaining weaker and weaker system characterizations.

A fundamental step of the approach is finding a suitable initial system characterization. For this step, we suggested the use of simulation. An alternative method is the technique proposed by Schiøler et al. [SJDL05], which relies on determining long-term rates of system streams.

Finally, we studied the precision of fixpoints in MPA. We showed that the abstraction loss experienced when analyzing systems with cyclic dependencies may be considerable, even for simple system architectures. In addition, we showed that it is very hard to predict the abstraction loss experienced in a fixpoint iteration. This was highlighted by a simple experiment, in which a small variation in the setup of a system severely affected the precision of the obtained fixpoint. How to effectively prevent the abstraction losses experienced with fixpoint iterations in MPA or SymTA/S remains an open issue.

4

Structured Event Streams

The MPA formalism assumes that all events or data token in a stream are homogeneous. However, this is often not the case in real systems. Complex stream-processing systems commonly contain streams that consist of event or data token of different types. In many cases, the particular type of a token determines not only the processing demand imposed on individual system components, but also how the token is routed through the network of components. In other words, events or data packets of different types may follow different processing paths in a distributed system. For instance, one can often observe the design pattern in which different event or data streams are merged into a joined stream which is processed by various system components, and then again split into individual sub-streams. Since such patterns cannot be captured in usual MPA models, one often faces serious abstraction losses when analyzing the performance of real systems. In this chapter, we extend the MPA framework such that it supports the modelling of such join/split scenarios. Specifically, we present two new methods that allow us to do so. The first method employs the abstract FIFO component introduced in Chapter 2, whereas the second method is based on Event Count Curves, an abstraction to represent the structure of a joined event stream.

4.1 Introduction

In distributed embedded systems that involve complex communication systems such as networks or buses, different event or data streams are often merged, transmitted over shared communication channels, and then separated again into individual streams. For example, data from different streams could be combined into frames that are transmitted over a network. In [RE08] such a combined stream is denoted as hierarchal event stream (HES). After the delivery of a frame, the data is unpacked, that is, the hierarchical data stream is split into its composing sub-streams. Another related scenario that does not involve packaging is illustrated in Figure 38. In this case, different event streams are simply merged to a single stream which is processed by an arbitrary complex subsystem, and then decomposed into individual sub-streams. There are many practical systems where these scenarios apply. For instance, they are encountered in the automotive domain in standards such as AUTOSAR or Flexray.

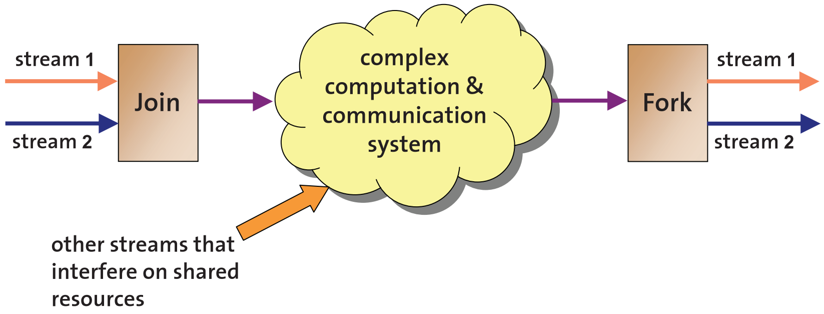

stream 1
stream 2
Join
complex computation & communication system
Fork
stream 1
stream 2
other streams that interfere on shared resources

Fig. 38: Joining and forking of streams

In this chapter, we introduce abstractions that permit us to take into account such scenarios in MPA. The main result is a method that allows us to join and fork abstract event streams with high accuracy without explicitly maintaining the types of individual events in the abstract stream models. The method is based on Event Count Curves, a model for the representation of structures in heterogeneous event streams. The concept of Event Count Curve is orthogonal to previously used stream representations such as PJD models and arrival curves. It is transparent to existing analysis components, and hence well suited to being embedded into analysis frameworks such as MPA or SymTA/S.

4.1.1 Motivational example

In this section, we describe a concrete distributed system in which different event streams are merged, processed and separated again. The simple system highlights the need for appropriate models in analytic performance analysis. It serves as example throughout the chapter, and is used to illustrate the presented methods. In Section 4.3.5 we provide the analysis of the example system and discuss the obtained results.

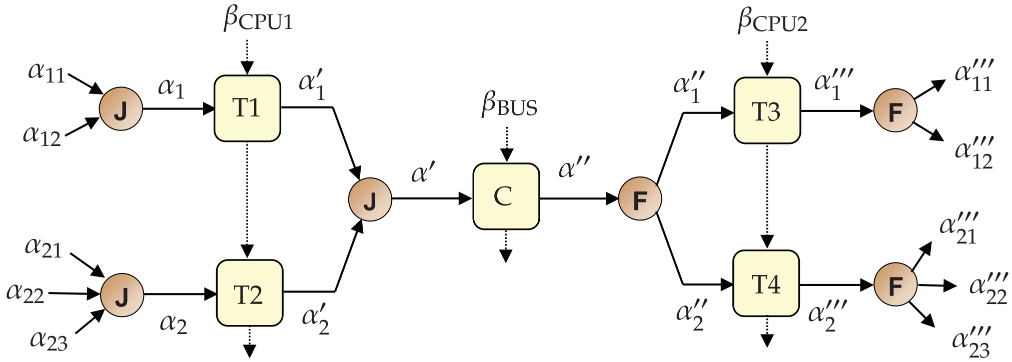


Fig. 39: MPA model of example system

The MPA model of the considered system is represented in Figure 39. It contains various event streams that are represented by arrival curves α. The arrival curves model the flow of data through the system. The system consists of several GPC components that represent computation or communication tasks. The service curves β model the amount of computation or communication resources available to the single tasks.

The system processes five input event streams α_{ij} that are joined (J) and forked (F) several times between the individual processing components. We assume 'OR' semantics for the join operator. This means that it produces one output event for each input event arriving on any of the inputs. We assume that the events of the joined stream are distinguishable with respect to their provenance. In particular, we say that an event is 'of type e_x' if it origins from the stream α_x. We also adopt this notation in the remainder of the chapter. Note that in the case of successive joins, the single events of a joined stream belong to several types. For instance, in the model shown in Figure 39, an event in the stream α' originating from stream α_{12} is of type e_1 but at the same time of type e_{12}. For the forking of streams, we assume that every input event is forwarded to only one output stream, depending on its type. In the figure, the indices of the event streams denote the processing paths followed by the various event types. For instance, the streams represented by α_{21} and α'''_{21} contain events of the same type.

We assume that both *CPU1* and *CPU2* implement preemptive fixed-priority scheduling, where task *T1* has higher priority than task *T2*, and task *T3* has higher priority than task *T4*. Further, we assume that all input event streams are periodic streams with jitter, and that the tasks are characterized by best-case and worst-case execution times (BCET, WCET). The corresponding parameters can be found in Table 3. The goal of the analysis is to characterize the output event streams as precisely as possible.

Stream	Period	Max. Jitter
α_{11}	100	30
α_{12}	90	15
α_{21}	30	0
α_{22}	80	20
α_{23}	75	5

Task	BCET	WCET
T1	2	3
T2	3	4
C1	9	12
T3	2	3
T4	4	5

Tab. 3: Parameters for the example system

4.1.2 Terminology

In this chapter, we distinguish *simple* and *structured event traces*. The former term refers to an event trace in which all events are of the same type. The latter is used for traces in which the events are of different types. A structured event trace results whenever several simple (or structured) event traces are joined, see Figure 40 for a graphical illustration. We also denote structured event traces as *joined event traces*.

We apply the same distinction to abstract event streams: A *simple event stream* is a set of simple event traces (assuming a unique event type for the different traces); a *structured event stream* is a set of structured event traces (assuming common event types for the different structured traces). A formal definition of these terms follows in Section 4.3.1. We also denote structured event streams as *joined event streams*.

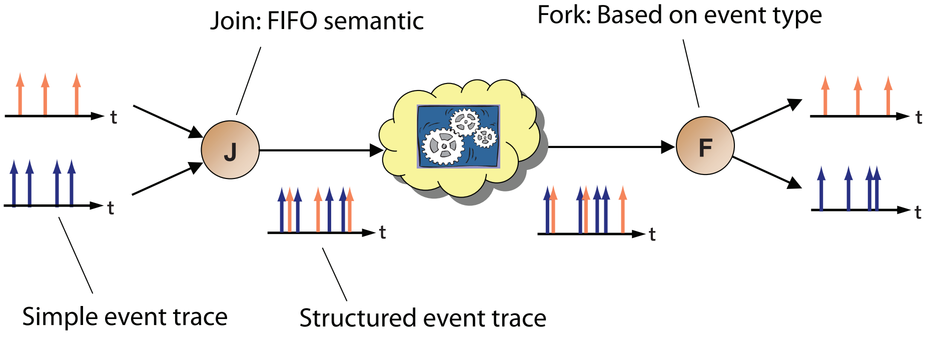


Fig. 40: Simple and structured event traces

4.1.3 Organization

This chapter is organized as follows. In Section 4.2 we briefly describe how the abstract FIFO component of the MPA framework can be used to handle joined event streams. In Section 4.3 we introduce the concept of Event Count Curves (ECC) and illustrate how it can be employed for abstractly modelling join and fork operations on event streams. Thereby, we distinguish two alternatives for the application of ECCs in complex systems (hierarchical vs. flat arrangement of ECCs). Further, we apply the proposed methods to the example system, and compare the obtained results. In Section 4.4 we describe a realistic case study. We conclude the chapter with a discussion of related work in Section 4.5, and a summary in Section 4.6.

4.2 FIFO Scheduling

A first alternative for modelling the processing of a joined event stream in MPA is to keep the individual sub-streams separated in the model, and to adapt the abstraction of the corresponding processing components such that they explicitly handle multiple input streams. To this end, we use the abstract FIFO component for the MPA framework introduced in Section 2.2.3.5. The use of the FIFO component for modelling the processing of a joined event stream is justified by the observation that the merging of two or more event traces preserves the arrival order of the individual events. In other words, processing the events of a joined stream by means of an abstract GPC component is semantically equivalent to processing the events of the various sub-streams with an abstract FIFO component. A simple example of this modelling principle is shown in Figure 41. On the left-hand side, the figure shows the processing of a joined event stream by a task T. After the processing the stream is forked into its composing sub-streams. On the right-hand side, the figure shows the equivalent model making use of a FIFO component. The component represents the processing of two equivalent tasks T that are scheduled in FIFO order of their activations.

The described modelling method represents structured streams as bundles of distinct streams. Hence, each time a system processes a structured stream, in the MPA model this has to be considered by abstracting the corresponding task by means of a FIFO component that processes a bundle of streams. This approach allows the modelling of arbitrary compositions and decompositions of event streams. However, it has a major drawback. It is not transparent to the existing abstract performance components of the MPA framework. In particular, it requires the explicit

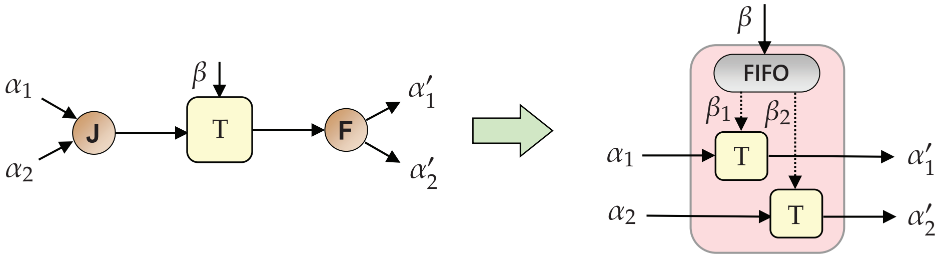


Fig. 41: FIFO model for the processing of a structured stream

adaptation of all the components of a model that process a structured event stream.

4.3 Event Count Curves

In this section, we introduce a second method for handling structured event streams that does not have the above limitation. The approach is based on the concept of Event Count Curves (ECC), a model that represents the structure of joined streams.

The method applies join and fork operations on abstract event streams to compose and decompose structured streams. It is based on the fact that the order of the events in a trace is preserved no matter how many and what kind of components process the trace. Hence, when merging different event streams into a structured one, we can store some information about the structure of the resulting stream, and then use this information at a later stage to split the structured stream again. Obviously, it is not wise to store the exact event type sequence for every possible trace of the structured stream, as this would result in an unbearable modelling overhead. However, for any sequence of events in a structured stream, we can still bound the number of events belonging to a given sub-stream, which is the main idea behind ECCs.

Note that the concept of ECCs is orthogonal to any event stream model such as PJD or arrival curves; event stream models describe the timing of event occurrences in event traces, whereas ECCs describe the occurrence of particular event types in sequences of heterogeneous events. Further, note that ECCs are involved in join/fork operations of event streams only; they are ignored (and also not affected) by the performance components of an MPA model. In other words, the abstraction of ECCs is totally transparent to all existing MPA component models, which means that the compositionality of the formalism is preserved.

4.3.1 Definitions

In this section, we formally define some terms that are relevant to the handling of structured event streams.

Def. 21: **(Structured event trace)** *A structured event trace with event types e_i, $i \in I$ is described by a set of arrival functions $r_i(s,t)$, $i \in I$. For some given times s and t with $s < t$, $r_i(s,t)$ denotes the number of events of type e_i that arrive in the time interval $[s,t)$.*

Note that, for a given a set of arrival functions $r_i(s,t)$, we can unequivocally determine the type and the timing of the individual events in the structured trace.

To characterize a structured event stream, we use a tuple of arrival curves that bound the total number of events which arrive in the stream and, in addition, a tuple of ECCs for each event type e_i. Intuitively, an ECC bounds the number of events that belong to a particular type within a given number of consecutive events of the structured stream.

Def. 22: **(Structured event stream)** *A structured event stream is characterized by a tuple of arrival curves $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$, $\Delta \geq 0$ and multiple tuples of ECCs $\gamma_i(n) = [\gamma_i^u(n), \gamma_i^l(n)]$, $n \geq 0$, one tuple for each event type e_i, $i \in I$. The upper and lower arrival curves bound the total number of events arriving in the structured stream, i.e., for any time interval $[s,t)$ with $s < t$, $i \in I$ we have*

$$\alpha^l(t-s) \leq \sum_{i \in I} r_i(s,t) \leq \alpha^u(t-s) \tag{4.1}$$

The upper and lower ECCs bound the number of events of type e_i within a certain number of consecutive events of the structured stream, i.e., for all $s < t$ and $i \in I$ we have

$$\gamma_i^l(\sum_{j \in I} r_j(s,t)) \leq r_i(s,t) \leq \gamma_i^u(\sum_{j \in I} r_j(s,t)) \tag{4.2}$$

For the single sub-streams that compose a structured stream, we can again define ordinary arrival curves.

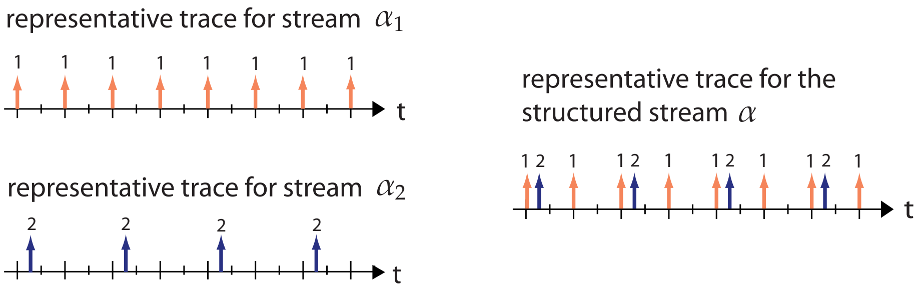
Def. 23: **(Arrival curves for sub-streams)** *For each event type e_i, $i \in I$, of a structured event stream, the arrival curves $\alpha_i(\Delta) = [\alpha_i^l(\Delta), \alpha_i^u(\Delta)]$, $\Delta \geq 0$, satisfy*

$$\alpha_i^l(t-s) \leq r_i(s,t) \leq \alpha_i^u(t-s) \tag{4.3}$$

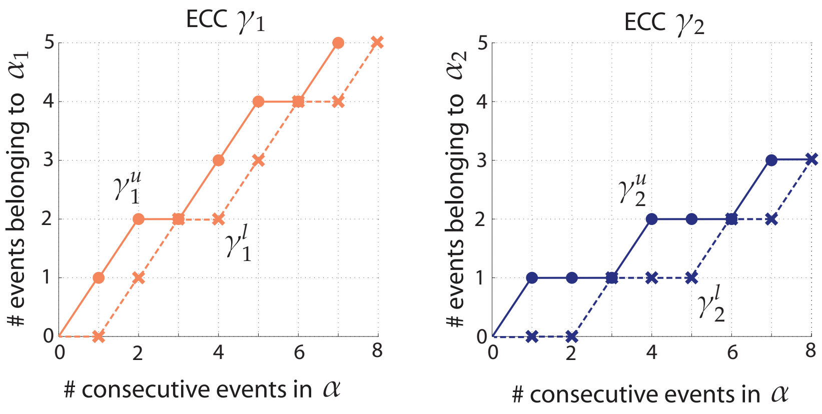
for all $s < t$.

Let us now illustrate the concept of ECCs by means of a simple example.

Ex. 1: *Consider two strict periodic event streams* α_1 *and* α_2 *with periods* $p1 = 10$ *and* $p2 = 20$*, respectively. Two example traces for the two streams are shown on the left side of Figure 42(a). Consider now the structured event stream* α *obtained by joining the simple event streams* α_1 *and* α_2*. A representative trace for* α *is shown on the right side of Figure 42(a). The ECCs* γ_1 *and* γ_2 *which describe the structure of the joined stream* α *are depicted in Figure 42(b). For any number of consecutive events in* α*,* γ_1 *and* γ_2 *specify bounds on the number of possible occurrences of events of type 1 and 2, respectively. For instance, for 5 consecutive events in the structured stream* α*, at least 3 and at most 4 events are of type 1, or in short,* $\gamma_1^l(5) = 3$ *and* $\gamma_1^u(5) = 4$*.*


representative trace for stream α_1
1 1 1 1 1 1 1 1
t
representative trace for stream α_2
2 2 2 2
t
representative trace for the structured stream α
1 2 1 1 2 1 1 2 1 1 2 1
t

(a) Event traces used in Example 1.



(b) ECCs for Example 1

Fig. 42: Representative event traces and ECCs for Example 1

Please note that ECCs are defined for integer values only. In Figure 42(b) the interconnecting lines are shown for illustration purposes only.

In the following sections we will also use the concept of pseudo-inverses of arrival curves and ECCs.

Def. 24: **(Pseudo-inverse of arrival curve)** *The pseudo-inverses of upper and lower arrival curves* $\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)]$ *are defined as*

$$\alpha^{-u}(n) = \inf\{\Delta \geq 0 : \alpha^u(\Delta) \geq n\} \tag{4.4}$$

$$\alpha^{-l}(n) = \sup\{\Delta \geq 0 : \alpha^l(\Delta) \leq n\} \tag{4.5}$$

Upper and lower arrival curves $\alpha^u(\Delta)$ and $\alpha^l(\Delta)$ denote the maximum and minimum number of events that may arrive in a stream in any time interval of length $\Delta \in \mathbb{R}^{\geq 0}$, respectively. Their pseudo-inverses have the following interpretation: $\alpha^{-u}(n)$ denotes the length of the shortest time interval in which there can be n event arrivals in the stream; $\alpha^{-l}(n)$ denotes the length of the longest time interval with n event arrivals.

Def. 25: **(Pseudo-inverse of ECC)** *The pseudo-inverses of upper and lower ECCs* $\gamma_i(n) = [\gamma_i^u(n), \gamma_i^l(n)]$ *are defined as*

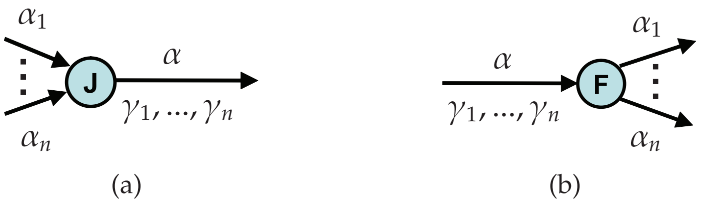
$$\gamma_i^{-u}(n_i) = \inf\{n \geq 0 : \gamma_i^u(n) \geq n_i\} \tag{4.6}$$

$$\gamma_i^{-l}(n_i) = \sup\{n \geq 0 : \gamma_i^l(n) \leq n_i\} \tag{4.7}$$

Upper and lower ECCs $\gamma_i^u(n)$ and $\gamma_i^l(n)$ denote the maximum and minimum number of events of type e_i in any sequence of $n \in \mathbb{N}$ events of the structured stream, respectively. Their pseudo-inverses are interpreted as follows: $\gamma_i^{-u}(n_i)$ denotes the minimum length of an event sequence that contains n_i events of type e_i; $\gamma_i^{-l}(n_i)$ denotes the maximum length of a sequence with n_i events of type e_i.

4.3.2 Join and Fork of Simple Event Streams

Let us now show how ECCs can be computed when several event streams are joined, and how they are utilized to split structured streams into sub-streams.



(a) (b)

Fig. 43: (a) Join operator for merging n event streams into one structured event stream. (b) Fork operator for decomposing a structured event stream into n sub-streams.

Consider first the fork operator shown in Figure 43(b). Given the joined event stream and the various ECCs that describe its structure, we can derive the individual sub-streams as follows.

Thm. 6: *Given is a structured event stream with arrival curve α and ECCs γ_i, $i \in I$. Then, we have*

$$\alpha_i^u(\Delta) = \gamma_i^u(\alpha^u(\Delta)) \tag{4.8}$$

$$\alpha_i^l(\Delta) = \gamma_i^l(\alpha^l(\Delta)) \tag{4.9}$$

Proof. The evaluation of $\alpha^u(\Delta)$ gives the maximum number n of events in the structured stream for a given interval Δ. This number can be translated using γ_i^u which, by definition, determines the maximum number of events of type e_i in a sequence of n events in the structured stream. The proof for $\alpha^l(\Delta)$ is analogous.

□

Consider now the join operator shown in Figure 43(a). Given the individual streams $\alpha_1, \cdots, \alpha_n$, we can compute the resulting structured event stream and the individual ECCs as follows.

Thm. 7: *Given are n event streams with arrival curves $\alpha_1, \cdots, \alpha_n$ that are joined to a single event stream. Then the resulting structured event stream is characterized by the arrival curves*

$$\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)] = \Big[\sum_i \alpha_i^u, \sum_i \alpha_i^l\Big]. \tag{4.10}$$

The ECCs of the structured event stream are determined by

$$\gamma_i(n) = [\gamma_i^u(n), \gamma_i^l(n)] = [\epsilon_i^{-l}(n), \epsilon_i^{-u}(n)] \tag{4.11}$$

with

$$\epsilon_i^l(n_i) = n_i + \sum_{j \neq i} \alpha_j^l(\alpha_i^{-u}(n_i)) \tag{4.12}$$

$$\epsilon_i^u(n_i) = n_i + \sum_{j \neq i} \alpha_j^u(\alpha_i^{-l}(n_i)) \tag{4.13}$$

Proof. The join operator forwards the events of all input event streams without delay. Thus, for any time interval we have that the number of event arrivals in the structured stream is equal to the sum of the event arrivals in the individual sub-streams, which justifies (4.10). Formally, we can prove (4.10) by combining the inequalities (4.1) and (4.3). For the computation of the ECCs, let us focus on $\gamma_i^l(n)$. The curve $\gamma_i^l(n)$ represents the minimum number n_i of events of type e_i in a sequence of n events of the structured stream. Consider now the pseudo-inverse of $\gamma_i^l(n)$ which we denote as $\epsilon_i^u(n_i)$. The curve $\epsilon_i^u(n_i)$ represents the maximum length n of an event sequence that contains n_i events of type e_i. The basic idea for finding this n for a given n_i is to construct a time interval in which there are n_i event arrivals of type e_i and as many event arrivals as possible for the remaining event types e_j with $j \neq i$. In (4.13) this is done by first considering the largest time interval Δ for which in sub-stream α_i there can be exactly n_i event arrivals, given by $\Delta = \alpha_i^{-l}(n_i)$. The total length n of the event sequence is then found by considering the maximum number of event arrivals of types e_j with $j \neq i$ in the interval Δ, given by $\sum_{j \neq i} \alpha_j^u(\Delta)$. We need to show that this length n corresponds to the maximum number of consecutive events in the structured stream, among which there are exactly n_i events of type e_i. In other words, we have to exclude the existence of an interval Δ' in which there n_i events of type e_i and *more* than n total events. By definition of α_i^{-l}, every interval Δ' with $\Delta' > \Delta$ contains more than n_i events of type e_i. Hence, such intervals can be ignored. On the other hand, reducing the size of Δ can definitely not increase the total length n of the considered event sequence. Thus, such an interval Δ' cannot exist, which proves the correctness of ϵ_i^u and consequently of γ_i^l. The derivation of γ_i^u is analogous.

□

Note that in terms of the overall arrival curve $\alpha(\Delta)$, the above described join operator is equivalent to the OR-composition of the input streams $\alpha_1, \cdots, \alpha_n$, as described in [HT07].

4.3.3 Hierarchical application of ECCs

In the presence of multiple successive join and fork operations, we can organize and apply ECCs in an hierarchical manner. Consider for instance the example system of Figure 39. We can model the merging and splitting of event streams by the operators introduced in Section 4.3.2. In particular, by joining the streams α_{11} and α_{12} we obtain the structured stream α_1 and two ECCs γ_{11} and γ_{12}. Similarly, the join of α_{21}, α_{22} and α_{23} results in the structured stream α_2 and the ECCs γ_{21}, γ_{22} and γ_{23}. The processed event streams α_1' and α_2' are then joined once more, which yields the structured stream α' and two ECCs γ_1 and γ_2. Following the various stream compositions, we can hierarchically organize the ECCs that describe the structure of the event stream α'. In particular, we can represent the hierarchy of event types by means of a tree, as shown in Figure 44. The edges of the tree represent ECCs that are computed when merging event streams, and applied when splitting event streams.

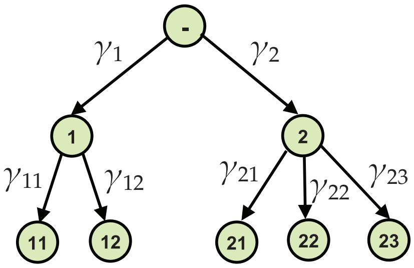


Fig. 44: Hierarchy of ECCs for the system of Figure 39

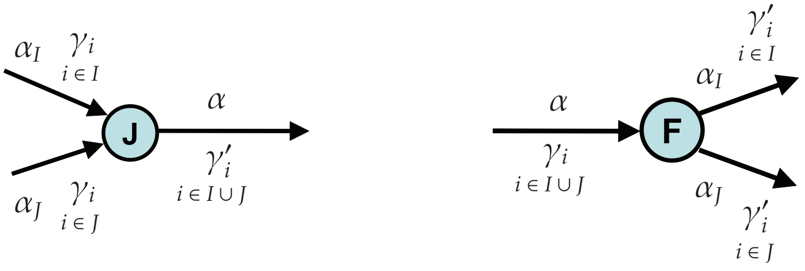
In the model of Figure 39, we apply the above hierarchy of ECCs to fork the event stream α'' into its composing sub-streams. More specifically, we first apply γ_1 and γ_2 to fork α'' into α''_{21} and α''_{22}. Afterwards, we fork α''_{21} and α''_{22} again by applying γ_{11}, γ_{12}, γ_{21}, γ_{22}, and γ_{23}.

4.3.4 Join and Fork of Structured Event Streams

The hierarchical organization of ECCs described above has a major drawback: a structured event stream can be decomposed only in the inverse order in which it has been composed. However, to extend the modelling scope of the method, it is highly desirable to provide join/fork operators that permit an *arbitrary* decomposition of structured streams into sub-streams, no matter how the structured streams were constructed. Consider again the motivational example introduced in Section 4.1.1. Assume a system that is analogous to the one of Figure 39, but in which the splitting of the event stream α'' is different. For instance, assume that the stream α_{21} is processed by task *T3*, instead of task *T4*. In other words, the composition and decomposition of the streams are not symmetrical. In this case, with the ECC model described so far, we have no means of correctly abstracting the system behaviour.

In this section we tackle the above problem, and introduce more general join and fork operators that operate on structured event streams. The new operators allow us to arbitrarily join and fork event streams, and rely on a flat hierarchy of ECCs. In other words, to characterize a structured stream, only ECCs referring to *simple* sub-streams are used. ECCs referring to *structured* sub-streams will not be computed and forwarded any longer.

Consider the join operator shown in Figure 45(a). It merges two structured event streams α_I and α_J. In contrast to the case of a hierarchical organization of ECCs, this join operator computes a new tuple γ'_i of ECCs for all *simple* sub-streams composing the outgoing structured stream. Hence, at any following component in the model, it is possible to isolate arbitrary subsets of the previously joined event streams. For this new join operator,



(a) (b)

Fig. 45: (a) Join operator to merge two structured event streams.
(b) Fork operator to split a structured event stream into two structured substreams.

the outgoing structured event stream can be characterized as follows.

Thm. 8: *Assume that a structured event stream with arrival curve α_I and ECCs γ_i, $i \in I$ is joined with a second structured event stream with arrival curve α_J and ECCs γ_i, $i \in J$ (see Figure 45(a)). Then, the resulting structured event stream is characterized by the arrival curve*

$$\alpha(\Delta) = [\alpha^u(\Delta), \alpha^l(\Delta)] = [\alpha_I^u(\Delta) + \alpha_J^u(\Delta), \alpha_I^l(\Delta) + \alpha_J^l(\Delta)]. \tag{4.14}$$

The ECCs of the resulting structured stream are given by

$$\gamma'_i(n) = [\gamma'^u_i(n), \gamma'^l_i(n)] = [\gamma_i^u(\gamma_I^u(n)), \gamma_i^l(\gamma_I^l(n))] \tag{4.15}$$

for $i \in I$. Here, we use the partial ECCs for stream I

$$\gamma_I^u(n) = \epsilon_I^{-l}(n) \tag{4.16}$$
$$\gamma_I^l(n) = \epsilon_I^{-u}(n) \tag{4.17}$$

with

$$\epsilon_I^l(n_I) = n_I + \alpha_J^l(\alpha_I^{-u}(n_I)) \tag{4.18}$$
$$\epsilon_I^u(n_I) = n_I + \alpha_J^u(\alpha_I^{-l}(n_I)) \tag{4.19}$$

The ECCs $\gamma'_i(n)$ for $i \in J$ are determined in analogous manner.

Proof. For any time interval, the number of event arrivals in the joined output stream is equal to the sum of the event arrivals in the two input streams, which justifies (4.14). For the outgoing ECCs, let us focus on $\gamma'^l_i(n)$ with $i \in I$. By ignoring that α_I and α_J are structured streams and by applying Theorem 7, we get (4.17). The ECC $\gamma_I^l(n)$ defines the minimum number of events of all types e_k with $k \in I$ in a sequence of n events of the structured stream α. In order to determine how many of these $\gamma_I^l(n)$ events are of a specific type e_i with $i \in I$, we just need to consider that α_I is a structured stream itself and apply the corresponding ECC γ_i^l. Hence, we obtain $\gamma'^l_i(n) = \gamma_i^l(\gamma_I^l(n))$. The derivation of $\gamma'^u_i(n)$ is analogous.

□

The above theorem can easily be extended to more than two inputs.

Let us now move to the fork operator shown in Figure 45(b). It splits a structured event stream α into two *structured* event streams α_I and α_J. The outgoing structured event streams can be characterized as follows.

Thm. 9: *Given is a structured event stream with arrival curve $\alpha(\Delta)$ and ECCs $\gamma_i(n)$, $i \in I \cup J$. Then, the arrival curve $\alpha_I(\Delta) = [\alpha_I^u(\Delta), \alpha_I^l(\Delta)]$ of a sub-stream I with event types e_i, $i \in I$, is characterized by by the bounds*

$$\alpha_I^u(\Delta) \leq \sum_{i \in I} \gamma_i^u(\alpha^u(\Delta)) \tag{4.20}$$

$$\alpha_I^u(\Delta) \leq \sup_{0 \leq \lambda \leq \Delta} \Big\{ \alpha^u(\lambda) - \sum_{i \in J} \gamma_i^l(\alpha^l(\lambda)) \Big\} \tag{4.21}$$

and

$$\alpha_I^l(\Delta) \geq \sum_{i \in I} \gamma_i^l(\alpha^l(\Delta)) \tag{4.22}$$

$$\alpha_I^l(\Delta) \geq \max\Big\{ \inf_{\lambda \geq \Delta} \Big\{ \alpha^l(\lambda) - \sum_{i \in J} \gamma_i^u(\alpha^u(\lambda)) \Big\}, 0 \Big\}. \tag{4.23}$$

The arrival curve $\alpha_J(\Delta)$ is derived in an analogous manner.
The ECCs $\gamma'_i(n) = [\gamma'^u_i(n), \gamma'^l_i(n)]$ with $i \in I$ are characterized by the bounds

$$\gamma'^u_i(n) \leq g_i^{-l}(n) \qquad \gamma'^u_i(n) \leq f_i^{-l}(n) \tag{4.24}$$

$$\gamma'^l_i(n) \geq g_i^{-u}(n) \qquad \gamma'^l_i(n) \geq f_i^{-u}(n) \tag{4.25}$$

with

$$g_i^l(n_i) = n_i + \sum_{k \in I \setminus \{i\}} \gamma_k^l(\gamma_i^{-u}(n_i)) \tag{4.26}$$

$$f_i^l(n_i) = \max\Big\{ \inf_{\lambda \geq n_i} \Big\{ \gamma_i^{-u}(\lambda) - \sum_{k \in J} \gamma_k^u(\gamma_i^{-l}(\lambda)) \Big\}, 0 \Big\} \tag{4.27}$$

$$g_i^u(n_i) = n_i + \sum_{k \in I \setminus \{i\}} \gamma_k^u(\gamma_i^{-l}(n_i)) \tag{4.28}$$

$$f_i^u(n_i) = \sup_{0 \leq \lambda \leq n_i} \Big\{ \gamma_i^{-l}(\lambda) - \sum_{k \in J} \gamma_k^l(\gamma_i^{-u}(\lambda)) \Big\} \tag{4.29}$$

The ECCs $\gamma'_i(n)$ with $i \in J$ are characterized in an analogous manner.

Proof (Sketch). Let us start with the upper bounds for $\alpha_I^u(\Delta)$. The curve $\alpha_I^u(\Delta)$ represents the maximum number of events that arrive on the upper output stream of Figure 45(b) in any interval of length Δ. This quantity can obviously not be larger than the maximum number of events that arrive on the input stream α within an interval of length Δ and that are of types e_i with $i \in I$. Hence, by applying Theorem 6 and summing up the obtained arrival curves α_i^u, $i \in I$, we get the bound (4.20). On the other hand, an alternative upper bound for $\alpha_I^u(\Delta)$ is found when subtracting from $\alpha^u(\Delta)$ the minimum number of events that are of types e_i with $i \in J$, which leads to (4.21). Note that in (4.21) we use a *sup* operator to guarantee the monotonicity of the resulting curve, as the difference of two arrival curves can in general be not monotone.[1] For $\alpha_I^l(\Delta)$ the reasoning is analogous. The only difference is that in (4.23) we also use a *max* operator to guarantee a positive result.

For the bounds on the outgoing ECCs, let us consider $\gamma'^l_i(n)$. As already done in Theorem 7, we determine such an ECC by deriving its pseudo-inverse which we call $\epsilon'^u_i(n_i)$. The function $\epsilon'^u_i(n_i)$ specifies the maximum length n of an event sequence in α_I that contains n_i events of a specific type e_i with $i \in I$. Similarly as for $\alpha_I^u(\Delta)$, we can again construct two alternative upper bounds for $\epsilon'^u_i(n_i)$ which we denote as $g_i^u(n_i)$ and $f_i^u(n_i)$, respectively. The function $g_i^u(n_i)$ computes the maximum number of events of all types e_h, $h \in I$, in any event sequence of the input stream α that contains exactly n_i events of the specific type e_i. The function $f_i^u(n_i)$ subtracts the minimum number of events of all types e_k, $k \in J$, that are present in any event sequence of the input stream α that contains n_i events of the specific type e_i. The ECCs $\gamma'^u_i(n)$ are determined in an analogous manner.

□

4.3.5 Experimental Evaluation

Let us now compare the quality of the discussed abstractions on the basis of the example system introduced in Section 4.1.1. In order to do so, we first build three new MPA models for the system of Figure 39. In the first model, we apply the method described in Section 4.2, that is, we use FIFO performance components with multiple inputs instead of the depicted GPC components. In the second model, we adopt hierarchies of ECCs as described in Section 4.3.3. In particular, we use the tree of Figure 44 to organize the ECCs that represent the structure of the streams α' and α''. The third model is based on a flat hierarchy of ECCs as described in Section 4.3.4. Thus, it contains two ECCs for α_1 and α'_1, five ECCs for α'

---

[1] The same construct was already used in Equation (2.23).

and α'', etc. Note that in the two latter models we can employ standard GPC components (cf. Figure 39) which are not affected by the structure of the input event stream.

We use the three models to bound the five output streams of the system. Figures 46 and 47 depict the results achieved for the output curves α_{12}''' and α_{21}'''. The figures show that the approach based on FIFO scheduling and the approach based on hierarchically organized ECCs provide close results. For the output stream α_{12}''', the FIFO approach determines tighter (i.e., less conservative) bounds. For the output stream α_{21}''', the approach based on hierarchical ECCs is tighter. The method based on the flat organization of ECCs provides the worst bounds for both output streams, with local exceptions as $\alpha_{21}'''^{\,u}$ shows. However, as pointed out earlier, this is the only method applicable in scenarios with unsymmetrical join and fork operations, and hence it is still highly useful in general.

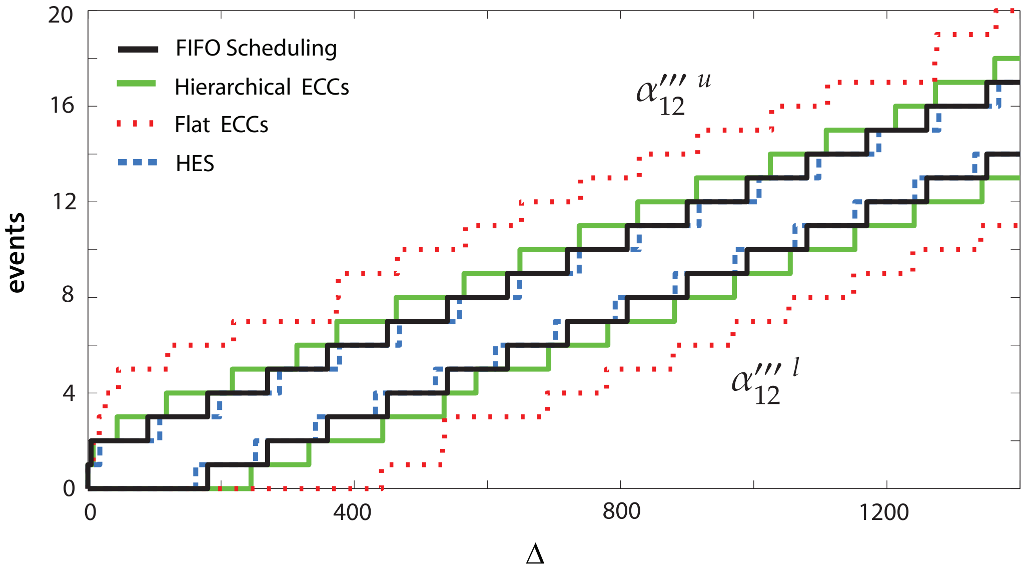


Fig. 46: Results for the characterization of the event stream α_{12}'''

We also compare the results of the proposed abstractions with the bounds obtained when applying the method described in [RE08]. The figures show that the hierarchical event model of [RE08] (HES) provides slightly better bounds than the approaches proposed in this chapter. Nevertheless, it has to be noted that, in contrast to the model of [RE08], the approaches based on ECCs have the advantage of not affecting the compositionality of existing analysis methods.

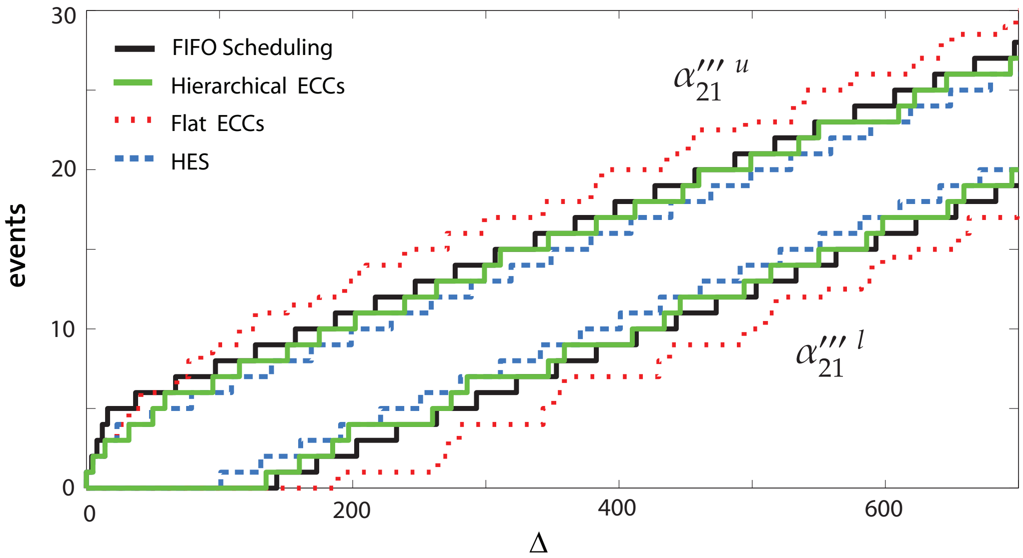


Fig. 47: Results for the characterization of the event stream α_{21}'''

4.4 Case Study

In this section we show how the presented theory can be applied to the analysis of a realistic distributed embedded system. We consider a heterogeneous communication system which is similar to the system described in Section 2.2.3.8, and which we denote as HCST in short. The HCST implements a distributed information/audio streaming application deployed in the waiting lounge of a large railway station.

4.4.1 General System Description

The HCST consists of various devices that are connected by a communication network. It comprises a central server (SERV), a backbone network, and a large number of end-devices (DEV). We assume that there are n network access controllers (NAC) connected in a chain along the backbone network, and that m distinct end-devices are connected to each NAC. Figure 48 shows the architecture of the HCST for the case $n = 3$, $m = 3$.

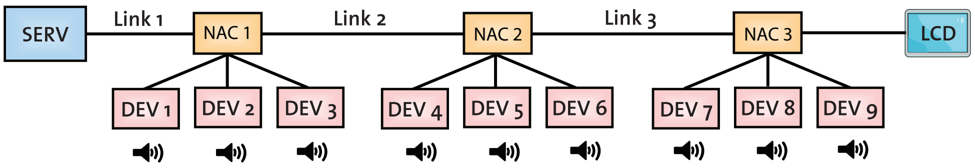


Fig. 48: System architecture

We assume that there is an end-device for each seat of the waiting lounge. The main function of the end-devices is the playback of audio streams that are transmitted over the backbone network by the server. We consider an on-demand audio system, where each traveler can choose their individual audio content from a large database stored on the server. We assume that the audio streams are transmitted in unicast mode, meaning that for each end-device there is a dedicated data stream from the server to the device. Besides on-demand audio streaming, the server executes a second application, namely the periodic broadcast of live train arrival/departure information (denoted as status data) over the backbone network. We assume that in the waiting lounge there are a number of LCD monitors connected to some of the NACs, displaying the transmitted train status data.

In this case study, we focus on the communication among the components of the HCST. In particular, we look at the timing of the data transmissions over the links of the network. We neglect the computations carried out by the components themselves, that is, we will not consider the execution times of the various processes on the server and the end-devices.

The HCST provides different QoS for the two different kinds of network traffic (audio streams and status data). We assume that, for the transmission of frames over the outgoing links, both the server and the NACs implement a preemptive fixed-priority arbitration policy, where audio traffic has higher priority than status traffic. This means that an ongoing transmission of status data over a link will be interrupted whenever there is an audio frame to be transmitted over the same link. The interrupted transmission will be resumed as soon as the link is free again. For the sake of simplicity, we assume that the transmission of frames can be interrupted and resumed at any time, without the need of retransmissions.

The goal of the analysis is to determine whether all frames with status data are guaranteed to reach their destination within a given deadline. To keep the illustration of the proposed methods simple, we restrict ourselves to the analysis of the small system architecture shown in Figure 48. In order to show meaningful effects still, we choose an accordingly small bandwidth for the backbone network.

4.4.2 Detailed specification

We consider a full duplex backbone network with a bandwidth of 5 Mbit/s. We are interested in the traffic from the server to the end-devices only. The server sends an individual audio stream to each of the nine devices shown in Figure 48. We index the audio streams with the number of the corresponding destination device. The considered audio streams have a net data rate of 384 kbit/s. The server partitions the data of each audio stream in frames of constant size. The sending pattern of the audio frames is periodic with a small jitter. The complete specification of the audio traffic is given in the upper part of Table 4.

	Size	**Period**	**Jitter**	**Deadline**
Audio Frames	1′518 Bytes	30 ms	5 ms	100 ms
Status Frames	106′500 Bytes	5 s	0 s	1.5 s

Tab. 4: Specification of the network traffic

The server also periodically sends status data to the LCD monitor. We assume that all the needed data is transmitted in a single frame of constant size. The detailed specification of the status data stream is reported in Table 4. The aim of the analysis is to clarify whether each status frame is guaranteed to reach its destination within the specified deadline.

4.4.3 Models and Analysis

Let us now proceed to the modelling of the specified system in MPA. We start from a rough model of the HCST in classical MPA in which the merging and splitting of event streams cannot be accurately captured. We then refine the MPA model by means of the newly proposed abstractions, and show how this permits us to drastically reduce abstraction losses.

The common basic abstraction of all the models is to represent the frame traffic in the network by means of timed event traces. This allows us to model communication components of the network such as data links by means of abstract components that process event traces. Figure 49 illustrates the basic modelling principle. The abstract processing components are triggered by incoming events which represent frames that need to be transmitted over the corresponding link. The processing time of an event in the abstract component corresponds to the transmission time of the frame on the concrete network link. The completion of a frame transmission is represented by the generation of an output event by the abstract component.

To model the transmission patterns of the server, we use Equations 2.13 and 2.14. This results in nine arrival curves $\alpha_1, \cdots, \alpha_9$, which represent the audio streams that are generated by the server. Similarly, we construct an arrival curve α_{status}, which models the status stream generated by the server. We use three service curves $\beta_1, \beta_2, \beta_3$ to represent the availability of the three network links for the transmission of frames. The fixed-priority arbitration policy is modelled by appropriately forwarding service curves

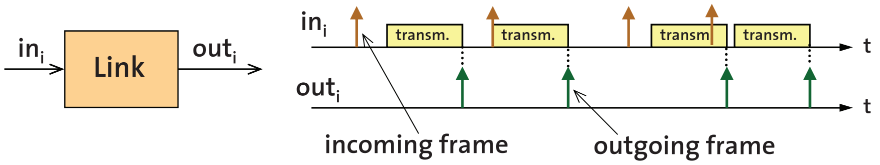


Fig. 49: Frame transmission abstracted as processing of timed event streams

among abstract processing components (cf. Figure 50). The transmission times for the frames on the network links are derived from the corresponding frame sizes and the network bandwidth. They amount to 2.429 ms for audio frames, and 170.4 ms for status frames.

Figure 50(a) shows a model of the HCST in classical MPA, that is, without the methods introduced in this chapter. The joining of the nine audio streams into a single stream is modelled by a simple sum of the corresponding arrival curves. This corresponds to a correct representation of the network traffic sent over link 1. However, in contrast to the newly proposed methods, in classical MPA there is no means to decompose the resulting stream again. Hence, we can only forward the entire joined event stream to other components in the model. In other words, with this model we overestimate the amount of network traffic on Links 2 and 3, and hence expect overly conservative performance results.

In Figure 50(b) we show the MPA model that makes use of the FIFO scheduling component as introduced in section 4.2. The models of the HCST that employ ECCs are shown in Figures 50(c) and 50(d). In the model of Figure 50(c), we consider a flat organization of the ECCs. Hence, whenever we have to branch off sub-streams from a structured stream, this can be done with a single fork operator as shown in the figure. In contrast, in the model of Figure 50(d) we join and fork sub-streams in a hierarchical manner.

4.4.4 Results

Table 5 sums up the results of the performance analysis. It reports the worst-case end-to-end delay for status frames predicted by the different models. For the sake of simplicity, we compute the end-to-end delay as sum of the response times of the individual links. Tighter bounds for the end-to-end delay could potentially be obtained by considering that a status frame cannot experience the worst-case interference of audio frames consecutively on all three links (cf. Section 2.2.3.6). However, such a holistic analysis is more involved and beyond the scope of this discussion.

The table shows that the newly proposed abstractions for modelling

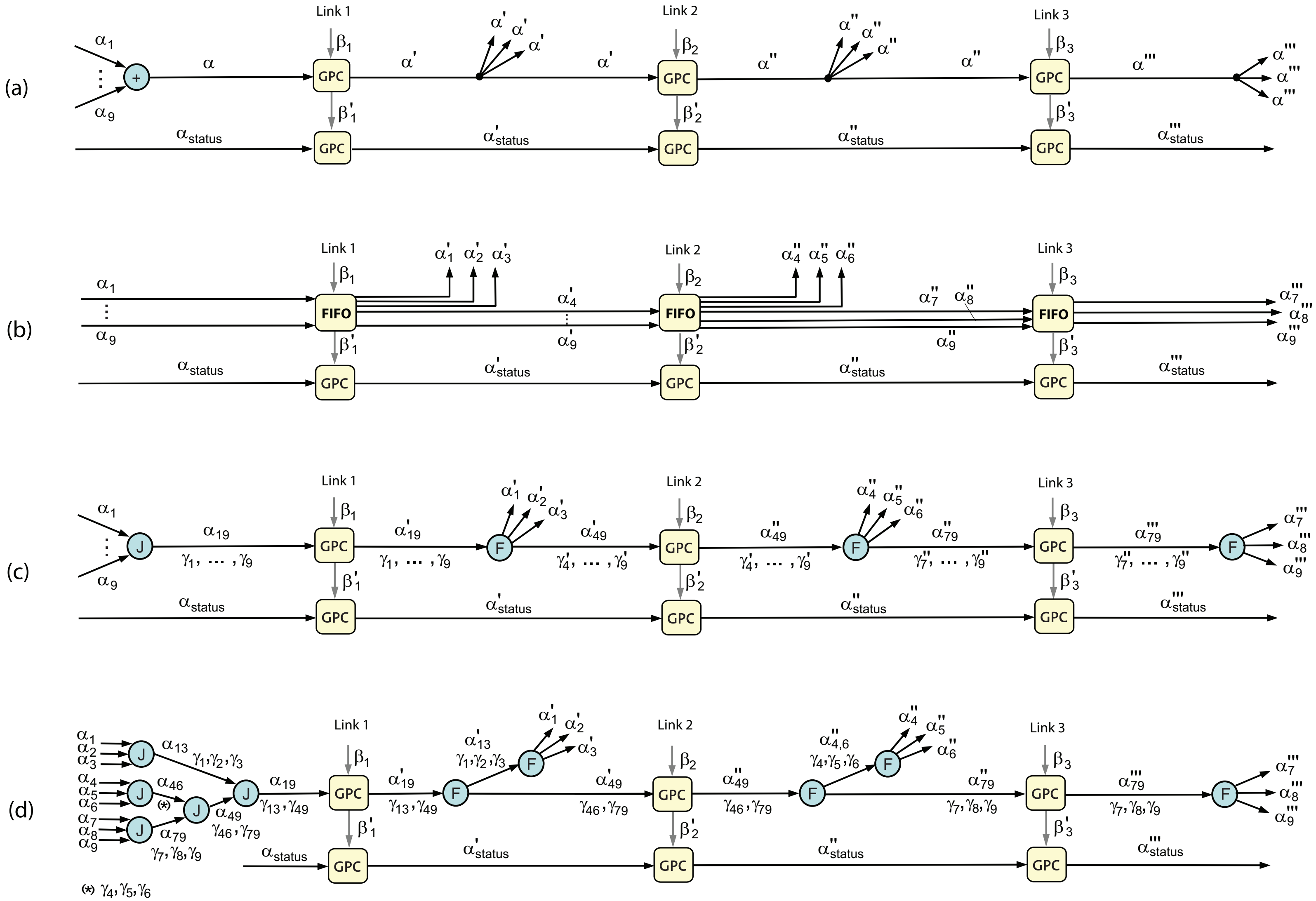

(a)
(b)
(c)
(d)
Link 1
Link 2
Link 3
GPC
FIFO
J
F
+
α_1
α_9
α
α'
α''
α'''
β_1
β_2
β_3
β'_1
β'_2
β'_3
α_{status}
α'_{status}
α''_{status}
α'''_{status}
α'_1 α'_2 α'_3
α'_4
α'_9
α''_4 α''_5 α''_6
α''_7 α''_8
α''_9
α'''_7 α'''_8 α'''_9
α_{19}
$\gamma_1, \dots, \gamma_9$
α'_{19}
α'_{49}
$\gamma'_4, \dots, \gamma'_9$
α''_{49}
α''_{79}
$\gamma''_7, \dots, \gamma''_9$
α'''_{79}
α_2 α_3 α_4 α_5 α_6 α_7 α_8
α_{13}
$\gamma_1, \gamma_2, \gamma_3$
α_{46}
(*)
α_{49}
γ_{46}, γ_{79}
α_{79}
$\gamma_7, \gamma_8, \gamma_9$
γ_{13}, γ_{49}
α'_{13}
$\alpha''_{4,6}$
$\gamma_4, \gamma_5, \gamma_6$
(*) $\gamma_4, \gamma_5, \gamma_6$

Fig. 50: MPA models of the HCST. (a) Classical (b) with FIFO components (c) with ECCs, flat (d) with ECCs, hierarchical

	Classical	**FIFO**	**ECC (flat)**	**ECC (hierar.)**
Max. delay	1.954 s	1.255 s	1.316 s	1.248 s

Tab. 5: Worst-case end-to-end delay for status frames derived with the different models

the merging and splitting of event streams lead to considerably better results for the analysis of the HCST compared to the naive modelling approach of Figure 50(a). In particular, based on the classical MPA analysis, we would have to reject the designed system, as we could not guarantee that all status frames arrive in time. On the other hand, the MPA models based on the abstract FIFO component or on ECCs show that the deadline for the transmission of status frames cannot be violated. Hence, the designed system fulfills the requirements. The reason for the better results is that the newly proposed methods permit us to capture the amount of audio traffic on Links 2 and 3 more precisely than the naive MPA model. This is confirmed by the better worst-case bounds on the availability of Links 2 and 3 to transmit status frames. For instance, Figure 51 shows the improvement for the lower service bound β'^{l}_{2} for Link 2. These improved service bounds finally yield tighter worst-case delay predictions.

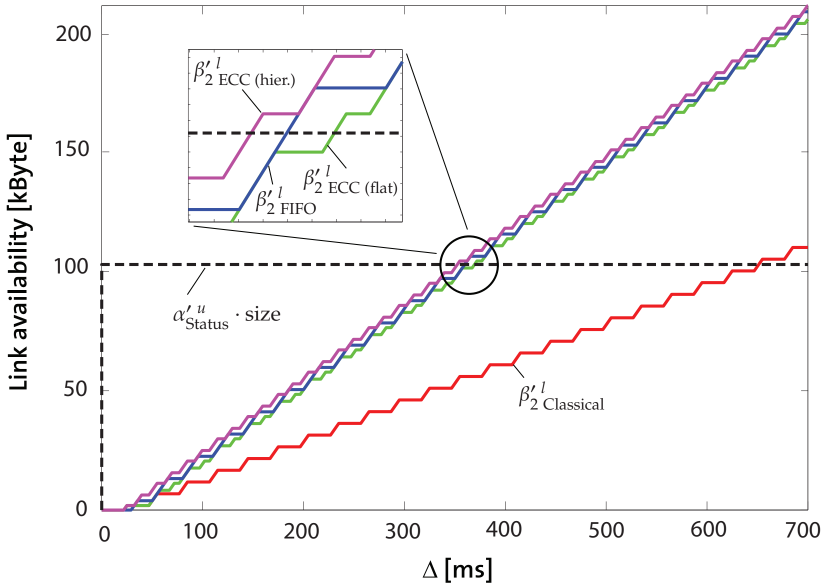


Fig. 51: Minimum availability of Link 2 for the transmission of status frames (β'^{l}_{2}) determined by the different models. The dashed line (α'^{u}_{status}) represents the maximum demand of link capacity.

Table 5 and Figure 51 show that the three newly proposed abstraction variants lead to close worst-case performance predictions for the system.

For the particular system under analysis, the model based on the hierarchical organization of ECCs turns out to be best. More importantly, however, the case study shows that all three proposed methods (FIFO, ECC hierarchical, ECC flat) permit us to reduce the abstraction loss experienced with classical MPA analysis considerably.

4.5 Related work

Rox et al. studied the merging and splitting of event streams in the context of SymTA/S. In [RE08] they proposed a hierarchical event model (HEM) to represent structured event streams which they call hierarchical event streams (HES). A hierarchical event model consists of an outer event model describing a joined event stream, and several inner event models describing the individual sub-streams. When a hierarchical event model is processed by a component, the outer event model is manipulated according to the usual SymTA/S transfer functions. In addition, appropriate functions update the various inner event models. The authors of [RE08] demonstrated significant reductions of abstraction losses when using HEMs instead of flat event stream models. Since the approach explicitly represents the individual sub-streams composing a joined stream, it allows the designer to arbitrarily decompose a joined stream whenever needed. However, the method is not transparent to existing component models. All SymTA/S modelling components need to be adapted to handle hierarchical event streams. Moreover, the analysis requires deep processing of the hierarchical event models.

Albers [ABS06] used a hierarchical data structure to describe repetitively occurring patterns within event streams. The approach enables fast scheduling analysis by ignoring lower levels of the event stream hierarchy. While in principle such a hierarchical event model can be obtained by combining several simpler event models, the approach of [ABS06] is different from the method discussed in this chapter. We do not assume any repetitive patterns for the occurrence of events, and our work is focused on the composition and decomposition of event streams.

FIFO scheduling has been considered in Network Calculus, see [LT01]. The results concern the service that is given to individual streams which are processed by some resource in FIFO order. These results are closely related to the transfer function of the abstract FIFO component described in Section 2.2.3.5.

The combination of multiple event streams for the activation of a task has been considered in both MPA and SymTA/S. For instance, the AND/OR conjunction of multiple event streams has been studied in [HT07, HHJ+05]. The OR-activation of tasks is closely related to the joining of event streams discussed in this chapter. However, the methods of [HT07, HHJ+05] do not support the separation of a joined stream into individual sub-streams.

Timing correlations in the presence of simple split-join scenarios of event streams have been studied in [HTSD07, SE09]. In contrast to the work discussed in this chapter, these methods are more limited since they do not consider different event types and cannot represent arbitrary merge and fork operations on streams.

Finally, we would like to note that Event Count Curves have also been considered in [WMT05] where they are called Type Rate Curves. However, in [WMT05] it is assumed that the type of an event determines only the workload imposed on the processing components of a system. We consider more general systems in which the type of an event determines its routing through the system. Therefore, we can represent the merging and forking of individual event streams.

4.6 Summary

In this chapter we introduced two novel methods for the modelling and analysis of joined event streams in MPA. The methods allow us to capture the structure of joined event streams that are processed by the components of a distributed system. The first approach is based on the abstract FIFO scheduling component introduced in Chapter 2. It keeps sub-streams separated in the system model, and represents structured event streams as bundles of individual event streams. Therefore, it ensures full flexibility regarding the splitting of joined streams. However, the approach requires the explicit adaptation of all the performance components of an MPA model. The second approach is based on Event Count Curves, an abstraction for representing the structure of joined event streams. This method explicitly handles the joining and forking of event streams, and is totally transparent to existing modelling components. Hence, it is highly suited for being embedded into frameworks for compositional performance evaluation such as MPA or SymTA/S. The main idea behind the Event Count Curves is to abstractly represent the structure of a joined event stream without explicitly maintaining exact sequences of typed events. We further extended the basic modelling approach based on Event Count Curves, such that arbitrary decompositions of event streams can be represented. This considerably extends the modelling scope of MPA. We evaluated and compared the proposed methods by means of experiments. The results indicate that none of the methods completely outperforms the other methods in terms of abstraction losses. In other words, the experiments highlight the utility of all the proposed approaches. Finally, we demonstrated the applicability of the described methods by analyzing a realistic application scenario.

A potential extension of the work described in this chapter is the consideration of join and fork operators with different semantics. For instance, one could consider general join/fork operators with multiple inputs and multiple outputs that join/fork event streams according to particular rules, e.g. load balancing.

5

Hybrid Performance Verification

A major cause for abstraction losses in MPA is the limited expressivity of the modelling approach. In fact, pessimistic performance predictions have to be expected whenever the MPA model of a system is correct (in the sense of conservative) but not accurate, meaning that it does not precisely reflect the real behaviour of the system. The problem is obviously not limited to MPA, but also concerns other analytical methods such as SymTA/S or MAST. In the following, we tackle this problem by proposing a novel hybrid approach to evaluate the performance of distributed real-time systems. The method abstracts system components either by an analytical and flow-based representation or by a state-based model in the form of timed automata. The interaction between the heterogeneous modelling components is captured by means of streams of discrete events. The resulting hybrid analysis framework enables trade-offs between analysis precision and efficiency. In particular, it allows the system designer to choose the level of detail at which individual system components are modelled. The proposed approach improves contemporary performance evaluation for the following reasons: (a) Abstraction losses as common for analytical methods can be reduced by means of more detailed system models; (b) State space explosion as common for state-based verification methods can be limited to the level of single system components. In this chapter, we first lay the theoretical foundations for the hybrid analysis framework. We then describe the implementation of a corresponding tool chain. The effectiveness of the method is demonstrated in a simple but realistic case study. Finally, we perform experiments to ascertain the scalability and the accuracy of the proposed approach.

5.1 Introduction

Distributed embedded systems often contain hardware or software components that exhibit complex, state-dependent behaviour. Typically, such behaviour cannot be precisely expressed by analytical component abstractions such as the ones employed in MPA. Examples of such state-dependent components are processing elements with caches as well as adaptive components with multiple execution modes, e.g., with dynamic power management. There exist several formal methods for verifying the behaviour of state-based systems. Finite state machines, Petri nets or Timed Automata are just a few examples of formalisms commonly used for this purpose.

Timed Automata (TA) [AD94] are a well-established formalism for the verification of real-time systems. Modelling and verification tools such as Uppaal [BLL+96] have considerably contributed to the dissemination of TA. The major advantage of TA is that they permit us to model complex real-time systems at an almost arbitrary level of detail. In model checkers such as Uppaal, the high-level model is then translated to a finite state-transition system which is used to verify the correctness of the model. Unfortunately, the state-transition system grows exponentially with the number of clocks and clock constants employed in the high level model. This fact heavily constrains the practical verification of large and complex real-time systems, and is commonly referred to as state space explosion problem. On the other hand, analytical (stateless) formalisms such as MPA, SymTA/S or methods from classical scheduling theory rely on the solution of closed-form expressions. Therefore, these approaches typically scale well with the size of models. In particular, this holds for compositional formalisms such as MPA or SymTA/S. However, there are two major disadvantages for these methods: (a) The evaluation of a system is limited to particular performance metrics such as worst-case latencies or backlogs; (b) The expressivity of these methods is limited to particular component models. Whenever the system under analysis contains complex components that cannot be accurately modelled in MPA, a conservative abstraction of the system has to be performed, which results in unprecise performance bounds (abstraction losses).

For overcoming the above deficiencies, we propose combining analytical performance evaluation and state-based system verification. Specifically, in this chapter we couple the formalism of Timed Automata (TA) with the framework for Modular Performance Analysis (MPA). We choose the former because it is widespread for the verification of state-based systems with real-time constraints. The latter is chosen as it is an advanced method for analytical performance evaluation. The result is a hybrid modelling framework in which for each individual system component the most appropriate formalism can be employed. The hybrid framework permits us to reduce the abstraction losses experienced with MPA. At the same time, it maintains scalability, as TA models are employed exclusively for those components where an MPA analysis is too pessimistic.

For coupling the formalisms of MPA and TA, the following obstacles have to be overcome:

- Unlike TA, the abstract MPA models lack concrete execution semantics.
- The verification of a TA model cannot be transformed to the evaluation of a closed-form expression.
- The two formalisms are based on different time domains. TA models operate on the conventional time line, whereas MPA models employ the time-interval domain.

We would like to highlight that the proposed methodology is not limited to MPA. The coupling of TA with other analytic formalisms such as SymTA/S, or any algorithm of classical scheduling theory, can be reduced to a special case of what is presented in this chapter.

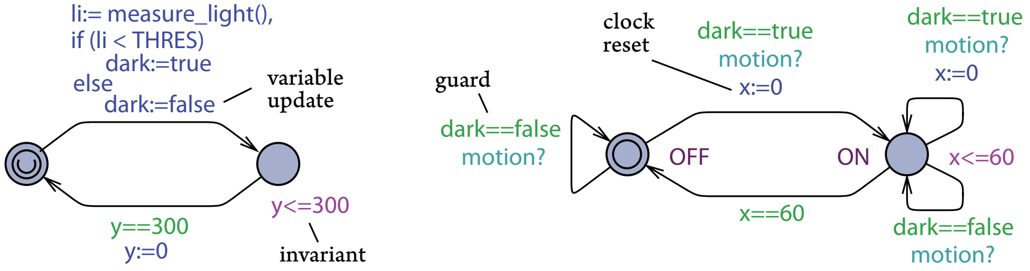
5.1.1 Organization

The content of this chapter is organized as follows. Section 5.2 gives a brief introduction to the modelling capabilities of TA. In Section 5.3 the general approach and the requirements for the hybrid analysis framework are described. In Section 5.4 we present a pattern that allows us to convert abstract stream representations such as PJD models or arrival curves to a network of TA. In Section 5.5, the inverse transformation is described, that is, how to extract abstract stream models from a TA-based subsystem. Section 5.6 contains a case study that highlights the usefulness of the approach. Moreover, it reports the results of experiments that investigate the scalability and the accuracy of the method. Finally, we discuss related work in Section 5.7 and provide a summary of the chapter in Section 5.8.

5.2 Timed Automata

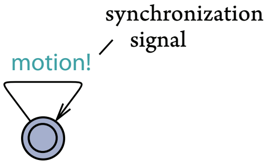
In this thesis we consider timed safety automata extended with variables as employed in the model checker Uppaal [Upp]. In the following we briefly summarize the modelling capabilities of such TA. Thereby, we focus on the most relevant aspects only. A more detailed introduction to the modelling and verification of TA networks can be found in [BY04].

A timed automaton is substantially a graph consisting of locations and edges. In addition, it may be equipped with clocks and (integer) variables. Clocks hold non-negative time values and model the advancing of time. More specifically, they track the amount of time that passed since their last reset. Three examples of TA are shown in Figure 52, which depicts the model of a smart outdoor light. The execution of the edges in a TA is controlled by conditions imposed on clocks and variables. Conditions related to locations are denoted as invariants whereas conditions related to edges are denoted as guards. The execution of an edge can only occur if both the guard of the edge and the invariant of the target location are satisfied. The execution of an edge may be followed by clock resets and/or variable updates. Note that the invariant of a target location must be satisfied *after* the clock resets and variable updates have taken place. A location can be defined as urgent, see e.g. the left location of the TA shown in Figure 52(a). In an urgent location, time cannot elapse. This means that once entered, an urgent location has to be left in zero time over one of the outgoing edges.



(a) Controller

(b) Light



(c) Environment

Local Declarations (Controller)

`clock` *y;*
`int` *li;*
`const int` *THRES = 100;*

Local Declarations (Light)

`clock` *x;*

System Declarations

`channel` *motion;*
`boolean` *dark;*

(d) Declarations

Fig. 52: Network of Timed Automata modelling a motion-sensing light

A key feature of the TA formalism is modularity. Complex systems can be represented by composing individual TA models. In such a network of TA, the interaction among the different modules relies on two concepts: (a) the synchronized execution of edges in distinct TA; (b) the use of shared global variables. For instance, the system model of Figure 52 consists of three individual TA. They represent a light, its controller and its environment, respectively. The modelled outdoor light is motion-sensitive. This means that it is equipped with a motion sensor and can automatically turn on when some movement is detected in its environment. We assume that the smart light also tracks the light intensity (li) of the environment. This operation is carried out every 300s. At the sensing of some motion, the light only turns on if the environment is dark, i.e., if the last measured light intensity was below a certain threshold. Once the light is turned on, it stays on until 60 seconds after the last registered movement. In the model of Figure 52, the two TA representing the controller and the light interact via the shared global variable *dark*. On the other hand, the TA models of the environment and the light interact via synchronized execution of their edges labeled with *motion*. Edges of different TA with the same synchronization label, also called channel, must be executed contemporaneously in an atomic manner. In Uppaal, two synchronizing TA are often denoted as sender and receiver. Uppaal supports two kinds of synchronization mechanisms:

- Binary synchronization
  Two TA, a sender and a receiver, jointly execute an edge with the same synchronization label (channel). In the sender the synchronization label is followed by an exclamation mark (cf. Figure 52(c)). In the receiver the synchronization label is followed by a question mark (cf. Figure 52(b)). In the case of binary synchronization, a sending edge of a TA can be executed only together with one corresponding receiving edge of another TA and viceversa.

- Broadcast synchronization
  If a synchronization label (channel) is defined as *broadcast*, the sender synchronizes with any number of receivers. In other words, one sending TA executes a sending edge, which can be understood as the emission of a signal, and between *0* and *n* receiving TA execute a receiving edge, which can be understood as the instantaneous reception of this broadcast signal. Note that even though the sender can synchronize with any number of receivers, participating in a broadcast synchronization is not facultative for receivers. In particular, all TA containing a receiving edge have to execute this edge if, at the time of the synchronization, it is executable, that is, if no conditions interdict its execution.

For better readability of the models, not all the TA shown in this thesis are syntactically correct with the respect to the syntax of Uppaal. In particular, when updating variables or resetting clocks, we sometimes use an `if-else` construct which in Uppaal has to be implemented by means of the ?-operator of ANSI C. The same applies to the max and min operators that we sometimes employ in assignments.

For formally verifying properties of a TA model, in Uppaal the high-level model is automatically translated to a finite state-transition graph. The complete state graph of a model is obtained by iteratively considering all possible edge executions in the high-level model until a fixpoint is reached. Properties of the modelled system, specified as propositions of simplified computation tree logic (CTL), are then verified by the model checker by systematically analyzing the resulting state graph. Such properties to be verified by the model checker are also called queries. For instance, we can use the query[1]

```
E<> (LIGHT.ON && not dark)
```

for verifying whether in the system of Figure 52 the light may be on when it is not dark. The result of the verification is that this property holds for the above system. The reason is that the TA representing the light may be in the location *ON* when the light intensity of the environment changes from dark to not dark.

5.3 General Approach and Requirements

The proposed hybrid analysis framework is based on a general scheme for interfacing abstract MPA components and TA models, see Figure 53. The main result of the scheme is the transformation of abstract event streams specified as arrival curves to sets of timed event traces specified as TA and vice versa. The scheme consists of two independent parts that we will discuss separately:

Input Interface

In Section 5.4 we show how to represent a general tuple of arrival curves by means of a network of TA called *input generator*. In Figure 53 this transformation is denoted with 'MPA → TA'.

Output Interface

In Section 5.5 we show how to extract a tuple of arrival curves from a TA model. In Figure 53 this transformation is denoted with 'TA → MPA'.

These two interfaces enable the evaluation of hybrid performance models in which the individual processing and communication components of a

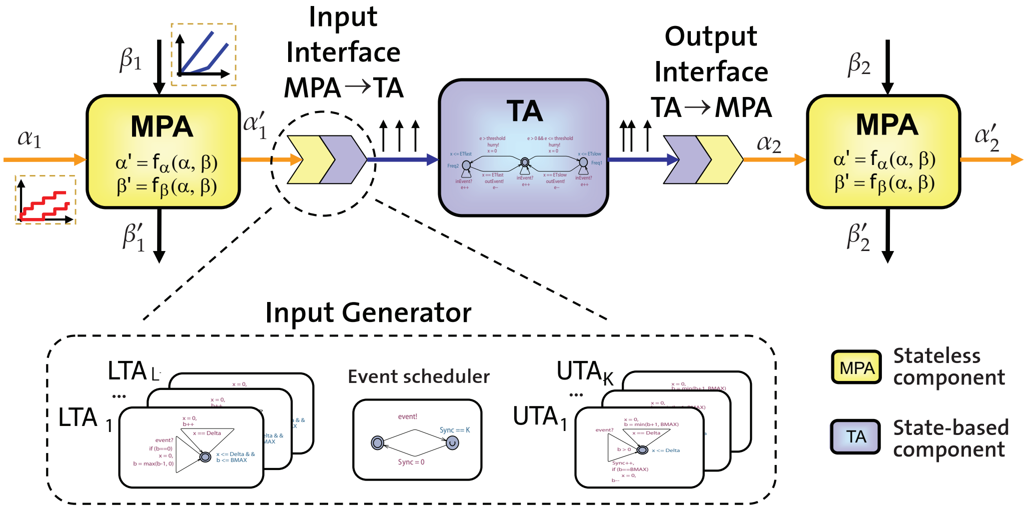


Fig. 53: Overview of hybrid analysis method

system are either abstracted on the basis of MPA or modelled by means of TA.

In this chapter we focus entirely on the conversion of arrival curves to TA and vice versa. The presented approach is, however, not limited to arrival curves, but can also be applied to service curves. The only limitation to consider is that in TA models, in particular when using Uppaal, one can only make use of discrete variables. Hence, we can only transform streams of discrete events or resource units, which on the level of RTC curves, refers to staircase functions.

Let us now discuss the requirements for the hybrid analysis framework. The main requirement is that the described input and output interfaces are *correct*, in the sense that they do not harm the safety of the analysis. In particular, in order to guarantee conservative analysis results, whenever transforming the abstraction of an event stream to another representation, we have to make sure that the conversion does not suppress any event trace of the stream.

R^{α_1}
Ω_1
R_1
R_2
Ω_2
R^{α_2}

(a) Interface MPA→TA

(b) Interface TA→MPA

Fig. 54: Interfaces of hybrid method seen as transformations among sets of event traces

Consider first a generic input interface (MPA→TA) in which a tuple of arrival curves α_1 is converted to an input generator, that is, to a network of

---

[1] In Uppaal `E<>` stands for 'possibly'.

TA. Figure 54(a) illustrates the corresponding transformation among sets of event traces where R^{α_1} denotes the set of event traces that conform to α_1 and R_1 represents the set of event traces specified by the input generator. We denote the set R_1 also as the set of event traces *producible* by the input generator. We say that the input interface is correct iff $R_1 \supseteq R^{\alpha_1}$. More precisely, for the input interface we require that

$$r \models \alpha_1 \;\Rightarrow\; r \in R_1 \qquad \forall r \in R. \tag{5.1}$$

Now consider a generic output interface (TA→MPA) in which the output of a TA subsystem is translated to a tuple of arrival curves α_2. Figure 54(b) shows the corresponding transformation, where R_2 denotes the set of traces producible by the TA subsystem and R^{α_2} represents the set of traces that conform to α_2. We say that the output interface is correct iff $R^{\alpha_2} \supseteq R_2$. More precisely, for the output interface we require that

$$r \in R_2 \;\Rightarrow\; r \models \alpha_2 \qquad \forall r \in R. \tag{5.2}$$

Note that the above properties guarantee a correct worst-case performance analysis, but do not exclude pessimistic analysis results. In particular, the accuracy of the performance analysis can degrade in cases in which $R_1 \supset R^{\alpha_1}$ holds for an input interface or $R^{\alpha_2} \supset R_2$ holds for an output interface. For instance, this corresponds to the case when the model of a system component is fed with more event traces than originally specified for the ingoing event stream. Such pessimistic transformations are avoided if $R_1 \subseteq R^{\alpha_1}$ holds for all input interfaces, and $R^{\alpha_2} \subseteq R_2$ holds for all output interfaces. However, in certain cases this requirement can be sacrificed, for instance to improve the efficiency of the analysis.

5.4 Interface MPA-TA

The input interface MPA→TA requires the conversion of an interval-based arrival curve $\alpha = [\alpha^u, \alpha^l]$ into a TA model that represents possibly infinitely many timed event traces. The basic principle for this conversion can be summarized as follows:

(i) Decomposition of the arrival curve into simpler curve components

(ii) Representation of each curve component as individual TA

(iii) Synchronization of the individual TA

The main idea behind the conversion of arrival curves into TA is the observation that any integer-valued arrival curve α can be represented by means of a set of linear staircase functions that are combined with minimum and maximum operations. By linear staircase function, we mean a step-function with uniform step-width, as the ones shown in Figure 55(a). In our approach, we decompose a general arrival curve α into its composing staircase functions and automatically translate each of them to an individual TA model according to a predefined modelling template. The resulting set of TA is then unified to a network in which the individual TA modules interact via synchronization and shared variables. Specifically, the synchronization of the TA reflects the minimum and maximum operations that combine the individual staircase functions. The resulting network of cooperating TA, shortly denoted as input generator, emits dedicated *event*-signals to the environment. These signals can be used for stimulating a user-defined TA model representing a processing or communication component of the system under evaluation. The input generator has to be constructed in such a way that each trace $r = (event, t1); (event, t2); ...$ of *event*-signals that it can emit conforms to the arrival curve α. In the following, we elaborate on the realization of the described conversion scheme. To keep the discussion simple, we first start with the simplest case, in which α^u and α^l consist of a single linear staircase function only. Thereafter, we consider the conversion of more general arrival curves.

5.4.1 Linear Pattern

We define upper and lower linear staircase functions as follows:

$$\alpha^{\{u,l\}}(\Delta) := N^{\{u,l\}} + \left\lfloor \frac{\Delta}{\delta^{\{u,l\}}} \right\rfloor \tag{5.3}$$

An example of two such curves is shown in Figure 55(a). The upper curve is defined by two parameters N^u and δ^u. The parameter N^u represents the maximum burst of the event stream, that is, the maximum number of events that may arrive at the same time. The parameter δ^u specifies the step-width of the upper arrival curve. Similarly, the lower curve is defined by two parameters N^l and δ^l. The parameter δ^l specifies the step-width of the lower arrival curve. The absolute value of the parameter N^l can be understood as the maximum number of consecutive δ^l delays that may separate two successive events. Hence, N^l indirectly specifies the longest possible interval without events in the stream. For both staircase curves α^u and α^l we assume a uniform step-height of one event.

5.4.1.1 Implementation

In this section we discuss how to encode the event stream specified by a tuple $\alpha = [\alpha^u, \alpha^l]$ of linear staircase functions in timed automata. As mentioned before, the goal is to construct an input generator in the form of a TA network that produces all the event traces specified by α. In our approach, we use two distinct TA to guard the conformance of the generated event traces to the upper and lower arrival curves. The upper bound α^u is guarded by a dedicated automaton that we call UTA. Similarly, the lower bound α^l is guarded by a dedicated automaton denoted as LTA. The binary synchronization of UTA and LTA ensures that the input generator produces only event traces r that fulfill $r \models \alpha$. The actual implementation of the input generator is shown in Figures 55(b) and 55(c). Both UTA and LTA employ their own counter b, clock x, as well as integer constants $N^{\{u,l\}}$ and $\delta^{\{u,l\}}$, see Figure 55(d) for the complete list of declarations, including the initialization of the counters. The two automata periodically increase their local counters. UTA does so every δ^u time units and LTA every δ^l time units. Both counters have a respective upper bound N^u and $|N^l|$. The two TA cooperate by synchronized execution of their *event*-edges. The non-deterministic emission of events is possible as long as for UTA $b > 0$ holds. On the other hand, an event has to be generated when both the local counter b and the clock x of LTA reach their respective thresholds $|N^l|$ and δ^l. This is enforced by a corresponding location invariant in LTA. When an event generation takes place, both UTA and LTA decrease their local counters b. It is also important to note that UTA resets its clock whenever its counter holds the *maximum* value N^u and an event is emitted, whereas LTA does so whenever its counter holds the *minimum* value 0 and an event is emitted.

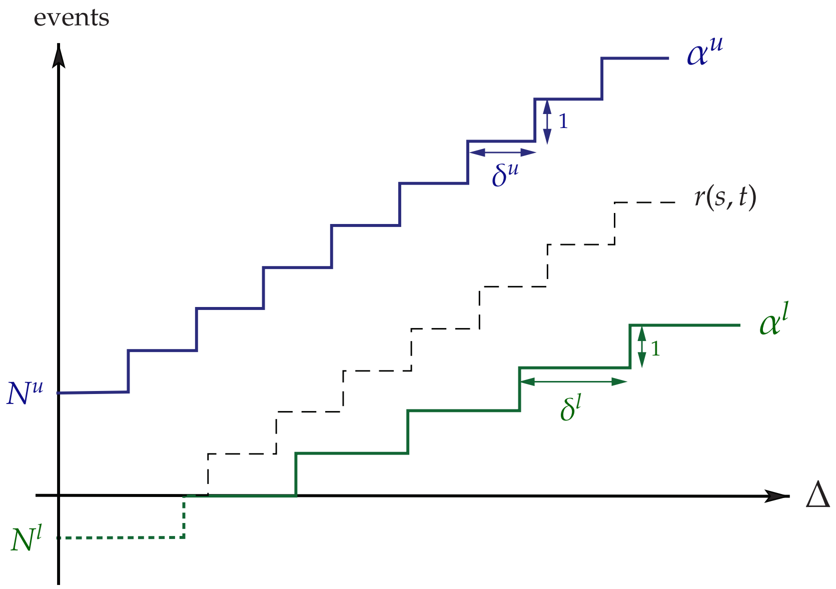
5.4.1.2 Correctness and tightness of interface

Let us now show that the input generator consisting of LTA and UTA is fully equivalent to the tuple of linear staircase curves $\alpha = [\alpha^u, \alpha^l]$ in terms of modelled event traces.

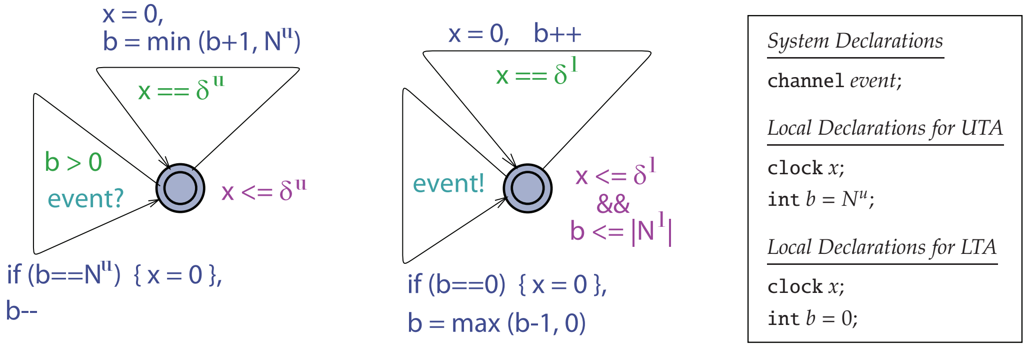
Thm. 10: *Let R^α be the set of event traces that conform to $\alpha = [\alpha^u, \alpha^l]$ as defined in Equation (5.3). Let R^{TA} denote the set of event traces producible by the input generator of Figure 55. Then, $R^\alpha = R^{TA}$.*

Proof (Sketch). To prove the theorem, we have to show that the described conversion is both correct and also tight. In the following, we briefly sketch the proof ideas for both properties for the linear input pattern. A more detailed proof can be found in Section 5.4.2, where we consider more complex input patterns.

- *Correctness* ($R^\alpha \subseteq R^{TA}$):
  We have to show that the event generator can produce *all* event traces r such that $r \models \alpha$. This can be shown by contradiction. Consider the upper bound α^u. Let us assume that there exists an event



(a) α^u and α^l modelled as linear staircase functions



System Declarations

```
channel event;
```

Local Declarations for UTA

```
clock x;
int b = N^u;
```

Local Declarations for LTA

```
clock x;
int b = 0;
```

(b) UTA for guarding α^u (c) LTA for guarding α^l (d) Declarations

Fig. 55: Linear pattern: Arrival curves and their TA-based implementation

trace r with $r \models \alpha^u$ that is not producible by the input generator. It follows that there is a time instant t at which UTA is blocked, meaning that its counter $b = 0$, but r contains a timed event $(event, t)$. However, by considering the prefix of any trace possibly produced by the input generator up to time t, it can be shown that an additional event at t would violate α^u, which contradicts the assumption $r \models \alpha^u$. Hence, such a trace r cannot exist. For the lower bound α^l the reasoning is analogous.

- *Tightness* ($R^\alpha \supseteq R^{TA}$):
  We have to show that the event generator can produce *only* event traces r such that $r \models \alpha$, that is, a general trace r can neither violate the upper bound α^u nor the lower bound α^l. These two statements can be justified as follows:

  (i) Upper bound
  UTA enforces that the input generator can only produce traces with at most $N^u + \lfloor \frac{\Delta}{\delta^u} \rfloor$ events in any interval of length Δ. This is because event emission is blocked once the counter b of UTA equals 0 and because the local clock x of UTA is reset at event emissions for which $b = N^u$ holds.

  (ii) Lower bound
  The invariant defined in LTA enforces that after at most $(N^l + 1) \cdot \delta^l$ time units of silence (no events), an event is emitted, and from then on a new event follows at least every δ^l time units. Therefore, every event trace produced by the input generator contains at least $\alpha^l(\Delta)$ events in any interval of length Δ.

□

5.4.2 Concave/Convex Pattern

In this section, we extend the input interface MPA→TA to more general arrival curves. We consider arrival curves that can be represented as the minimum or maximum of multiple linear staircase functions. Specifically, we consider arrival curves of the form

$$\alpha^u(\Delta) := \min_i \left\{ \alpha_i^u(\Delta) \right\} \tag{5.4}$$

$$\alpha^l(\Delta) := \max_i \left\{ 0,\ \alpha_i^l(\Delta) \right\} \tag{5.5}$$

where the curves $\alpha_i^{\{u,l\}}$ are defined according to (5.3), each with an individual pair of parameters $N_i^{\{u,l\}}$, $\delta_i^{\{u,l\}}$. An example of such a tuple of arrival curves is shown in Figure 56(a). We require that the following constraints hold for the parameters of the individual staircase curves $\alpha_i^{\{u,l\}}$:

$$\begin{array}{lcll} N_i^u & < & N_j^u & \forall i < j \\ |N_i^l| & < & |N_j^l| & \forall i < j \\ \delta_i^{\{l,u\}} & > & 0 & \forall i \\ \delta_i^l & > & \delta_j^l & \forall i < j \\ \delta_i^u & < & \delta_j^u & \forall i < j \end{array} \tag{5.6}$$

The above constraints ensure that each linear staircase function $\alpha_i^{\{u,l\}}$ contributes to the overall minimum/maximum derived in (5.4)/(5.5). In other words, they exclude redundant specifications of curve components. In the context of our work, we denote upper arrival curves that satisfy (5.4) and (5.6) as *concave*. Similarly, we say that a lower arrival curve is *convex* if it satisfies (5.5) and (5.6). In the hybrid analysis framework, we employ the above pattern for specifying concave/convex approximations of general arrival curves. These approximations are a key factor for ensuring simple and scalable TA models.

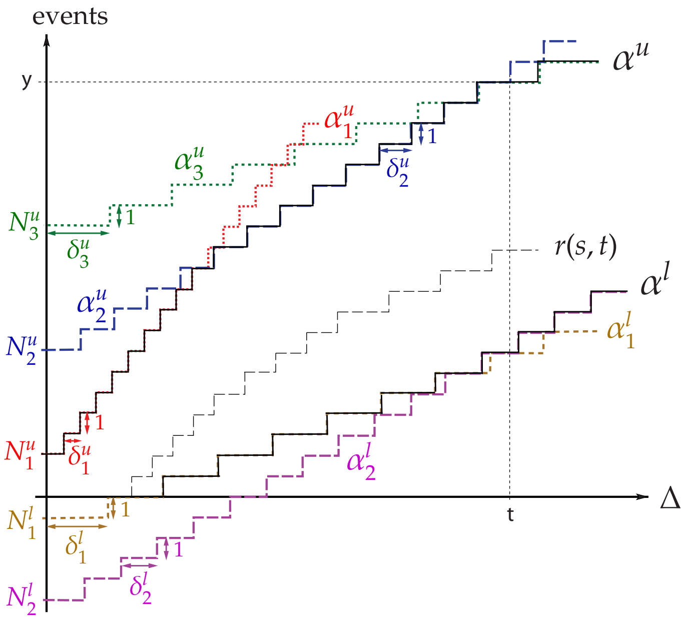
5.4.2.1 Implementation

Let us now discuss how to represent an event stream specified by a tuple $\alpha = [\alpha^u, \alpha^l]$ of concave/convex arrival curves by means of Timed Automata. The basic principle is to construct an input generator consisting of several UTA and LTA, each of which guards a linear staircase function $\alpha_i^{\{u,l\}}$ that composes the overall arrival curve α. The individual automata of the input generator need to cooperate in a way that reflects the minimum and maximum conditions of Equations 5.4 and 5.5. Specifically, these equations translate to the following requirements:

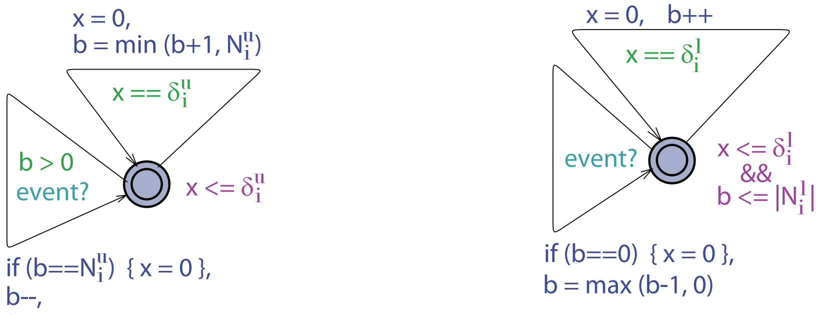
1. **Minimum condition**
   At a generic time point t, the input generator **may** emit an event if the resulting event stream conforms to *all* staircase functions α_i^u.

2. **Maximum condition**
   At a generic time point t, the input generator **has to** emit an event if not emitting it would lead to an immediate violation of *at least one* staircase function α_i^l.

The actual TA implementation of the input generator for concave/convex arrival curves is shown in Figure 56. The automata of Figures 56(b) and 56(c) are templates of TA models. They are instantiated multiple times according to the number of requested UTA and LTA. Each of these instances has its own local counter b_i, clock x_i, as well as integer constants

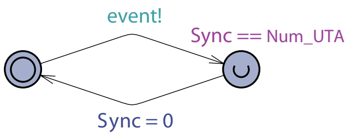


(a) α^u and α^l modelled as minimum and maximum of linear staircase functions



(b) UTA for guarding α_i^u

(c) LTA for guarding α_i^l



Local Declarations for UTA i

`clock` *x*;
`int` *b* = N_i^u;

Local Declarations for LTA i

`clock` *x*;
`int` *b* = 0;

System Declarations

`broadcast channel` *event*;
`int` *Sync* = 0;

(d) Scheduler for emitting events

(e) Declarations

Fig. 56: Convex/concave pattern: Arrival curves and their TA-based implementation

$N_i^{\{u,l\}}$ and $\delta_i^{\{u,l\}}$, where the index i identifies the particular instance of UTA or LTA.[2] As before, each UTA and LTA periodically increases its local counter b_i every $\delta_i^{\{u,l\}}$ time units and reduces it whenever an event is generated. The major difference to the TA network of Section 5.4.1 is that now we employ broadcast synchronization to coordinate the various automata to generate events. Specifically, the input generator contains a dedicated scheduler automaton, represented in Figure 56(d), that initiates the generation of events by sending a synchronization signal over the broadcast channel *event*. All UTA and LTA of the network receive this signal and participate in the event generation by executing their corresponding *event*-edges. However, the generation of an event is only possible if the minimum condition holds. On the other hand, the emission of an event must be enforced when the maximum condition applies. In the TA network of Figure 56, these two conditions are implemented as follows:

1. The **minimum condition** is enforced by the location invariant $Sync = Num\_UTA$ defined in the scheduler automaton (see Figure 56(d)). The invariant ensures that the target location of the *event*-edge in the scheduler can be entered only if the global variable *Sync* is equal to the number of UTA in the network. Since at a broadcast synchronization each UTA increments *Sync* by 1, we have that the synchronization takes place only if *all* UTA can participate.[3] In other words, the input generator may emit an event only if all UTA allow to do so.

2. The **maximum condition** is implemented by means of the location invariants of the different LTA. A *single* LTA enforces an event generation whenever executing the *event*-edge is the only way for circumventing the violation of the invariant.

The described implementation of the minimum condition yields full synchronization of all the TA in the network, meaning that either all *event*-edges are jointly executed or none. This also comports the nice feature that the input generator deadlocks if the represented upper and lower arrival curves are not consistent. For concave/convex arrival curves as described above, this is the case if the upper and lower curve cross each other. Hence, by verifying the deadlock-freeness of the input generator, we can exclude inconsistent tuples of arrival curves.

---

[2] For the sake of better readability, we omit the index i for local variables and clocks in the automata of Figures 56(b) and 56(c).

[3] Remember that the invariant of a target location must be satisfied *after* the variable updates have taken place.

5.4.2.2 Correctness and Tightness of Interface

As done for the linear pattern, we have to show that the conversion mechanism is correct and tight. In particular, we have to prove that $R^{\alpha} = R^{TA}$ also holds for concave/convex arrival curves. This will be done in two steps. In step (A) we prove that the input generator can produce all the event traces that conform to $\alpha^{\{u,l\}}$, i.e., $R^{\alpha} \subseteq R^{TA}$. In step (B) we prove that the input generator cannot violate the upper and lower bounds $\alpha^{\{u,l\}}$, i.e., $R^{\alpha} \supseteq R^{TA}$. The final result is summarized in (C).

(A) Correctness ($R^{\alpha} \subseteq R^{TA}$)

Lem. 1: *Let R^{α^u} be the set of event traces that conform to α^u defined according to Equations (5.4) and (5.6). Let R^{UTA} be the set of event traces producible by the input generator of Figure 56 without LTA, shortly denoted as Gen^{UTA}. Then, $R^{\alpha^u} \subseteq R^{UTA}$.*

Proof. This will be shown by contradiction. Let us assume that there is an event trace r with $r \models \alpha^u$, but that is not producible by the input generator Gen^{UTA}. This means that Gen^{UTA} can reproduce the event trace r only up to a time t_x at which the generator blocks even though r contains a timed event $(event, t_x)$. Let us denote this partially reproduced trace as r^{TA}. Then, we have $(event, t_x) \in r$ but $(event, t_x) \notin r^{TA}$. Since Gen^{UTA} is blocked at time t_x, there must be at least one UTA that prevents event generation at t_x, that is, $\exists b_i : b_i(t_x) = 0$, otherwise Gen^{UTA} could generate an event. Let t_j with $t_j \leq t_x$ be the earliest point and t_k with $t_k > t_x$ be the latest point in the trace r such that in $b_i(t) = 0$ holds $\forall t \in [t_j, t_k)$, see also Figure 57. In other words, t_k is the next time when counter b_i is incremented and $[t_j, t_k)$ represents the blocking interval of Gen^{UTA} due to UTA_i. Let t_1 with $t_1 < t_x$ be the latest point in the trace r where $b_i(t_1) = N_i^u$ was satisfied. Under these assumptions, we conclude that for the number of events $r(t_1, t_2)$ that are present in the trace r in the interval $[t_1, t_2)$ with $t_2 \in (t_x, t_k)$ we have

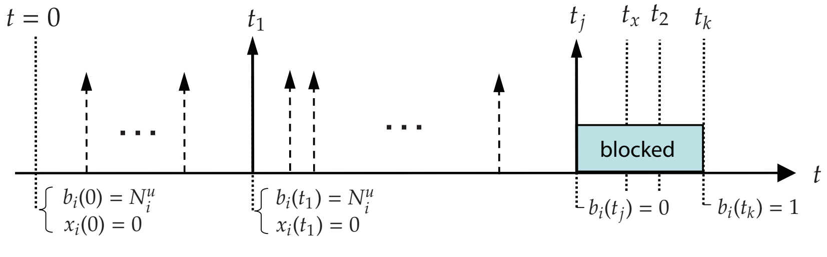


Fig. 57: Timed event trace r^{TA} with evaluation of b_i and x_i

$$r(t_1, t_2) > N_i^u + \left\lfloor \frac{(t_2 - t_1)}{\delta_i^u} \right\rfloor = \alpha_i^u(t_2 - t_1), \tag{5.7}$$

because r has at least one additional event with respect to r^{TA}, namely the event $(event, t_x)$. Obviously, this number of events violates the bound imposed by α_i^u. Thus such a trace r cannot exist.

□

Lem. 2: *Let R^{α^l} be the set of event traces that conform to α^l defined according to Equations (5.5) and (5.6). Let R^{LTA} denote the set of event traces producible by the input generator of Figure 56 without UTA, shortly denoted as Gen^{LTA}. Then, $R^{\alpha^l} \subseteq R^{LTA}$.*

Proof (Sketch). Analogously to the proof of Lemma 1, this can be shown by contradiction. We assume that there is a trace r with $r \models \alpha^l$, but that is not producible by the generator Gen^{LTA}. In particular, we assume that for a given time interval $[t_1, t_2)$ with $\Delta = t_2 - t_1$ the trace r contains less events than the minimum number of events enforced by Gen^{LTA} for any time interval of length Δ. By reasoning about the behaviour of Gen^{LTA} for different interval sizes (see also proof of Lemma 5), it can be shown that such a trace r does not exist.

□

Lem. 3: *Let R^{α} be the set of event traces that conform to $\alpha = [\alpha^u, \alpha^l]$ defined according to Equations (5.4), (5.5) and (5.6). Let R^{TA} denote the set of event traces producible by the input generator of Figure 56. Then, $R^{\alpha} \subseteq R^{TA}$.*

Proof. Follows directly from Lemmata 1 and 2.

□

(B) Tightness ($R^{\alpha} \supseteq R^{TA}$)

Lem. 4: *Let R^{TA} denote the set of event traces producible by the input generator of Figure 56. Let α^u be defined according to Equations (5.4) and (5.6). Then, $r \models \alpha^u \;\; \forall r \in R^{TA}$.*

Proof (Sketch). As discussed above, the parameters N_i^u and δ_i^u of UTA_i correspond to the parameters of the respective linear staircase function α_i^u. It is easy to see that UTA_i permits the generation of at most $N_i^u + \left\lfloor \frac{\Delta}{\delta_i^u} \right\rfloor$ events and that with $b_i = 0$ it blocks event production. The minimum condition as defined above gives that for any $\Delta \in \mathbb{R}^{\geq 0}$ the maximum number of producible events is bounded by $\min_i(N_i^u + \left\lfloor \frac{\Delta}{\delta_i^u} \right\rfloor)$. This is exactly what was defined for α^u in Equation 5.4 and 5.6.

□

By way of illustration, consider again Figure 56(a). If y events are produced in a time interval of length t, then the counter $b_3 = 0$ and UTA_3 blocks the event generation. From then on, α_3^u determines the amount of producible events, as it is the minimal curve among the upper staircase functions.

Lem. 5: *Let R^{TA} denote the set of event traces producible by the input generator of Figure 56. Let α^l be defined according to Equations (5.5) and (5.6). Then, $r \models \alpha^l \;\; \forall r \in R^{TA}$.*

Proof (Sketch). As discussed above, the parameters N_i^l and δ_i^l of LTA_i correspond to the parameters of the respective linear staircase function α_i^l. It is easy to see that LTA_i enforces the generation of an event after at most $(N_i^l + 1) \cdot \delta_i^l$ time units of silence, followed by further event enforcements every δ^l time units. Hence, for each stream $r \in R^{TA}$ we have $r \models \alpha_i^l$. Since the same must hold for all remaining staircase functions α_j^l with $j \neq i$, we also have $r \models \max_k \left\{ \alpha_k^l \right\} = \alpha^l$. It remains to show that each event enforced by some LTA can effectively be generated. Without loss of generality, we assume that the input generator is deadlock-free, meaning that the upper and lower bounds imposed by the various UTA and LTA are consistent. This implies that the events enforced by some LTA can never be suppressed by an UTA. Hence, the input generator cannot violate the bound α^l.

□

Lem. 6: *Let R^α be the set of event traces that conform to $\alpha = [\alpha^u, \alpha^l]$ defined according to Equations (5.4), (5.5) and (5.6). Let R^{TA} denote the set of event traces producible by the input generator of Figure 56. Then, $R^\alpha \supseteq R^{TA}$.*

Proof. Follows directly from Lemmata 4 and 5.

□

(C) Identity ($R^\alpha = R^{TA}$)

Thm. 11: *Let R^α be the set of event traces that conform to $\alpha = [\alpha^u, \alpha^l]$ defined according to Equations (5.4), (5.5) and (5.6). Let R^{TA} denote the set of event traces producible by the input generator of Figure 56. Then, $R^\alpha = R^{TA}$.*

Proof. Follows directly from Lemmata 3 and 6.

□

5.4.3 Extensions

In practice, systems may show event streams that do not adhere to the above described input pattern. While in such cases the pattern can still be used to conservatively approximate the arrival curves, it is often desirable to avoid approximations. Therefore, in the following, we briefly sketch two possible refinements for the discussed input pattern.

5.4.3.1 Shifted staircase functions

In the input pattern described in Sections 5.4.1 and 5.4.2, each linear staircase function has a uniform step width for all steps. However, in practice one often encounters staircase functions $\hat{\alpha}^{\{u,l\}}$ that have an initial offset $\theta^{\{u,l\}}$, meaning that they are horizontally shifted. For instance, this is the case for the arrival curves of a pjd event stream. An example of such translated curves is shown in Figure 58.

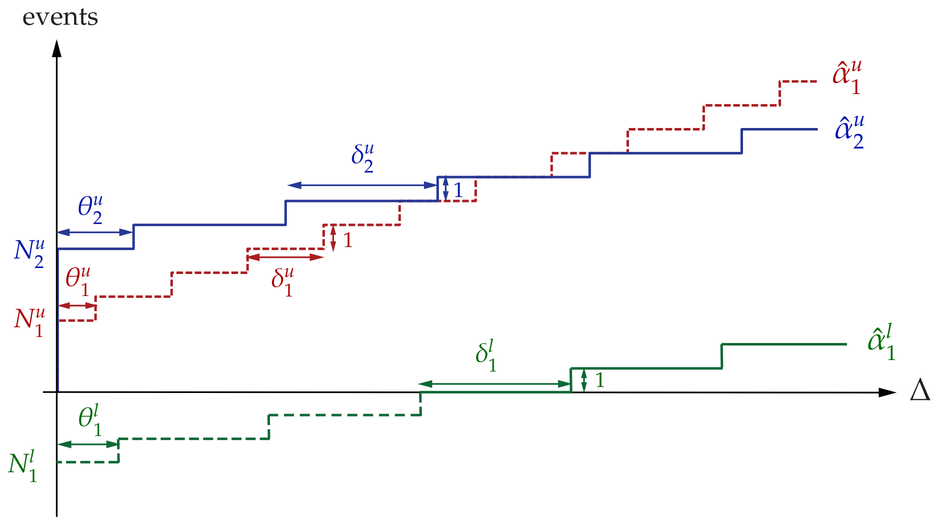


Fig. 58: Shifted staircase functions with offset $\theta_i^{\{u,l\}}$

Staircase functions with initial offset can be modelled by a more general version of the TA shown in Figure 56. In the following, we will explain the underlying principle of the model on the basis of $\hat{\alpha}_i^u$. The corresponding automaton $U\hat{T}A_i$ is shown in Figure 59. There are two differences with respect to UTA_i:

1. At each event generation the counter b_i is decreased by e_i^u units.
2. Scaled constants $\hat{N}_i^u$ and $\hat{\delta}_i^u$ are used as counter threshold and increment period.

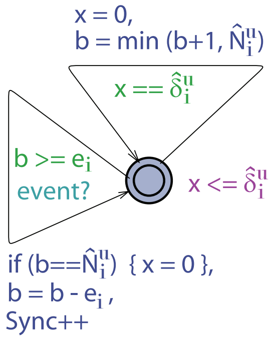


Fig. 59: TA for shifted upper staircase function $\hat{\alpha}_i^u$

The idea for achieving the initial offset θ_i^u in the modelled curve is to use a value for e_i^u that is not a factor of $\hat{N}_i^u$. In this way, after an immediate generation of the maximum burst of events, the next event can be generated earlier compared to the previous model. This is because after the generation of the maximum burst, the counter will be equal to a residual value $0 < b_i < e_i^u$ and, hence, the generation threshold e_i^u is reached earlier. The following equations permit to derive the model parameters e_i^u, $\hat{N}_i^u$, and $\hat{\delta}_i^u$ for given curve parameters θ_i^u, N_i^u, and δ_i^u:

$$\hat{\delta}_i^u = \gcd(\delta_i^u, \theta_i^u); \qquad e_i^u = \frac{\delta_i^u}{\hat{\delta}_i^u}; \qquad \hat{N}_i^u = (N_i^u + 1) \cdot e_i^u - \frac{\theta_i^u}{\hat{\delta}_i^u} \tag{5.8}$$

Consider, for instance, $\hat{\alpha}_2^u$ shown in Figure 58 and defined by $N_2^u = 6$, $\delta_2^u = 8$, and $\theta_2^u = 4$. By applying Eq. 5.8 we obtain $\hat{\delta}_2^u = 4$, $e_2^u = 2$, and $\hat{N}_2^u = 13$. One can easily verify that with these parameters the automaton of Figure 59 generates event traces which conform to $\hat{\alpha}_2^u$.

For lower curves $\hat{\alpha}_i^l$ the reasoning is analogous and results in the following equations:

$$\hat{\delta}_i^l = \gcd(\delta_i^l, \theta_i^l); \qquad e_i^l = \frac{\delta_i^l}{\hat{\delta}_i^l}; \qquad \hat{N}_i^l = N_i^l \cdot e_i^l - \frac{\theta_i^l}{\hat{\delta}_i^l} + 1 \tag{5.9}$$

5.4.3.2 Non-concave/convex patterns

Another issue is that, in practice, systems may sometimes not show strictly concave or convex patterns. For instance, the overall upper input curve may have parts with decreasing step widths (see, e.g., $\alpha'^u_{2,\mathrm{MPA}}$ in Figure 68), or the lower curve may contain parts with increasing ones. Assuming that the non-concave (non-convex) patterns occur only finitely often within an upper (lower) arrival curve, one can handle the situation by making use of nested sets of UTA, LTA, and local synchronization.

This strategy permits us to represent local minima and maxima and, hence, non-concave and non-convex arrival curves. Specifically, in order to implement such patterns, we need to encapsulate the respective sets of co-operating UTA and LTA in their own sub-system. These subsystems can be implemented analogously to the pattern illustrated in Section 5.4.2, but requiring slightly adapted TA-specifications with respect to the employed thresholds.

Note that in the case of non-concave/convex input curves, the specification of an upper and a lower bound might be inconsistent even if the two curves do not intersect. This problem is described in more detail in [AM10]. In our framework, such inconsistencies can easily be detected by model checking a corresponding deadlock query.

5.4.4 Limiting the Complexity

The complexity of model checking TA is exponentially bounded by the number of clocks and clock constants [AD94]. Thus, it is straightforward to see that the efficiency of the approach is closely related to the number of linear staircase functions employed to model lower and upper input curves. In the following, we propose a simple method that permits us to approximate a general arrival curve with the concave/convex combination of just a few linear staircase functions. The approach first approximates the arrival curve by a PJD event model, and then derives the parameters for the corresponding staircase curves $\alpha_i^{\{u,l\}}$.

Arrival curves are, in general, more expressive than PJD models. However, every arrival curve can be conservatively approximated by a PJD model. Given a general arrival curve to be fed into a TA-based component, we first use the algorithm described in [KHET07] to approximate it with a PJD model. Then, we convert the PJD parameters to a set of appropriate parameters $N_i^{\{u,l\}}$ and $\delta_i^{\{u,l\}}$. Finally, we use these parameters to specify the input generator for the TA-based component as described in Section 5.4.2.

The upper arrival curve of a PJD model can be represented by the minimum of at most two linear staircase functions α_1^u and α_2^u. Specifically, two linear staircase functions are needed if $d > 0 \;\wedge\; d > p - j$, while only one linear staircase function suffices otherwise. For the lower bound of a PJD model, one linear staircase function α^l is always sufficient. The parameters of the staircase functions are computed as follows:

- Case $d = 0 \;\vee\; d \leq p - j$:
  $N^u := \left\lceil \frac{j}{p} \right\rceil + 1; \qquad N^l := -\left\lceil \frac{j}{p} \right\rceil; \qquad \delta^u := \delta^l := p$

- Case $d > 0 \wedge d > p - j$:
  $N_1^u := 1; \quad \delta_1^u := d; \quad N_2^u := \left\lceil \frac{j}{p} \right\rceil + 1; \quad N^l := -\left\lceil \frac{j}{p} \right\rceil; \quad \delta_2^u := \delta^l := p$

Note that an exact representation of a PJD model by means of linear staircase functions α_1^u, α_2^u and α^l is not always possible if we exclude horizontally shifted staircase functions. However, in such a case the above formulae guarantee a correct (i.e. conservative) approximation of the PJD model. On the other hand, if we use the generalized input model described in Section 5.4.3.1, then we can precisely represent any PJD model by means of at most three linear staircase functions. In this case, the parameters of the staircase functions are computed as follows:

- Case $d = 0 \vee d \leq p - j$:
  $\hat{N}^u = x \in \mathbb{N}^+,\ e^u = y \in \mathbb{N}^+$ such that $\frac{x}{y} = \frac{j}{p} + 1 \ \wedge \ \gcd(x, y) = 1$;
  $\hat{\delta}^u = \frac{p}{e^u}$
  $\hat{N}^l = v \in \mathbb{N}^+,\ e^l = w \in \mathbb{N}^+$ such that $\frac{v}{w} = \frac{j}{p} \ \wedge \ \gcd(v, w) = 1$;
  $\hat{\delta}^l = \frac{p}{e^l}$

- Case $d > 0 \wedge d > p - j$:
  $\hat{N}_1^u = 1; \quad e_1^u = 1; \quad \hat{\delta}_1^u = d$
  $\hat{N}_2^u = x \in \mathbb{N}^+,\ e_2^u = y \in \mathbb{N}^+$ such that $\frac{x}{y} = \frac{j}{p} + 1 \ \wedge \ \gcd(x, y) = 1$;
  $\hat{\delta}_2^u = \frac{p}{e_2^u}$
  $\hat{N}^l = v \in \mathbb{N}^+,\ e^l = w \in \mathbb{N}^+$ such that $\frac{v}{w} = \frac{j}{p} \ \wedge \ \gcd(v, w) = 1$;
  $\hat{\delta}^l = \frac{p}{e^l}$

The approximation of arrival curves with PJD models represents a simple way to coarsely bound an event stream with few staircase functions. However, in the presented hybrid analysis approach the interface MPA→TA is of course not limited to PJD curves. Any other algorithm that correctly bounds a general arrival curve with an arbitrary number of linear staircase functions $\alpha_i^{\{u,l\}}$ can be used as interface between the two domains.

5.5 Interface TA-MPA

In this section we describe the realization of the output interface TA→MPA. The goal is to bound the output of a TA subsystem by means of a tuple of arrival curves $\alpha' = [\alpha'^u, \alpha'^l]$. As described in Section 5.3, the requirement for a correct output interface is $R^\alpha \supseteq R^{TA}$, where R^{TA} denotes the set of event traces producible by the TA subsystem. In other words, the output of the TA subsystem may be approximated, as long as the approximation is conservative. The main concept used for constructing valid output curves $\alpha'^{\{u,l\}}$ can be considered just the reverse of the event generation: We derive a set of linear staircase functions α'^{u}_{i} and α'^{l}_{i} for the output of the TA subsystem, and construct an overall output curve $\alpha'^{\{u,l\}}$ by means of minimum and maximum operators. To achieve this goal, we couple the system under analysis (including the input generator) with a set of observing TA. The model checking of reachability queries for these TA-systems allows us to derive the parameters $N^{\{u,l\}}_{i}$ and $\delta^{\{u,l\}}_{i}$ that uniquely characterize α'^{u}_{i} and α'^{l}_{i}. In the following, we first describe the TA that are used for verifying individual staircase parameters. After that, we describe the overall composition strategy for constructing a valid output curve $\alpha'^{\{u,l\}}(\Delta)$.

5.5.1 Observer TA

We use the observer TA shown in Figure 60 to derive various characteristics of the output event stream.

(a) *Maximum burst size:* An upper bound for the maximum number of events that the TA system can generate simultaneously can be verified by means of the observer automaton of Figure 60(a) and the query[4]: `A[] (count<=estimate)`.

(b) *Maximal distance between two successively emitted events:* We can verify a bound on the maximum pause time between two output events by employing the observer of Figure 60(b) and the query `A[] (pause imply x<=estimate)`.

(c) *Arbitrary upper staircase function α'^{u}_{i}:* For obtaining an individual upper staircase function, we employ the observer TA of Figure 60(c) which witnesses the violation or invulnerability of the respective curve. The witnessing TA moves into the location `violation` once the respective curve is violated, that is, once the observed system produces too many events. The reachability of this location is ascertained by the query `A[] (not violation)`. Hence, given some staircase parameters N^{u}_{i} and δ^{u}_{i}, we can determine whether the corresponding staircase function is a valid upper bound for the output event stream.

(d) *Arbitrary lower staircase function α'^{l}_{i}:* To obtain an individual lower staircase function, we employ the observer TA of Figure 60(d) and use the same principle as described above: Given some parameters

---

[4] In Uppaal `A[]` stands for 'always invariantly'.

N_i^l and δ_i^l, we can determine whether the corresponding staircase function is a valid lower bound for the output event stream.

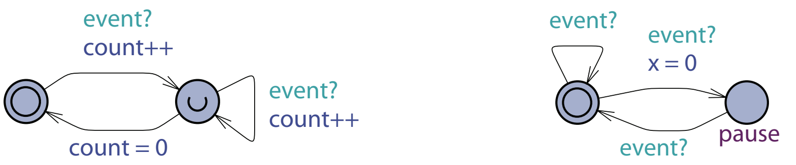
(e) *Long-term rates:* In order to construct output curves $\alpha'^{\{u,l\}}$ that approximate the system behaviour well also for large time intervals, we need to make sure that we follow the long-term event output rate. By long-term rate of an arrival curve α we mean the inverse of the limit $\lim_{\Delta\to\infty} \dfrac{\alpha(\Delta)}{\Delta}$, which always exists as detailed in [JPTY08]. The largest δ_i^u and the smallest δ_i^l of *any* correct upper and lower output staircase function denote upper and lower bounds on the long-term rate of the output. The principle of efficiently verifying that a given staircase function represents this upper or lower bound will be explained by means of $\alpha_i'^u$. The procedure for $\alpha_i'^l$ is analogous and is omitted for conciseness. The idea is to verify whether the observed system can produce an event trace such that for arbitrary long intervals the rate of the trace is not slower than δ_i^u. To do so one may employ the TA depicted in Figure 60(e). This TA monitors the difference between the number of event arrivals allowed by the rate δ_i^u and the number of events actually produced by the observed system. Once this difference exceeds a constant `D`, the TA moves to the location `drift`. If there is a trace for which the observer TA stays indefinitely in the location `count`, it means that we have found a trace that, on the long-term, never gets slower than δ_i^u, i.e., the rate δ_i^u is not overly pessimistic for the system output. Such a trace can be found as counterexample to the query: `count --> drift`.[5]

The above automata can be employed in many different ways to derive arrival curves $\alpha'^{\{u,l\}}$ for the output event stream. In the following list, we summarize only a few of the possibilities:

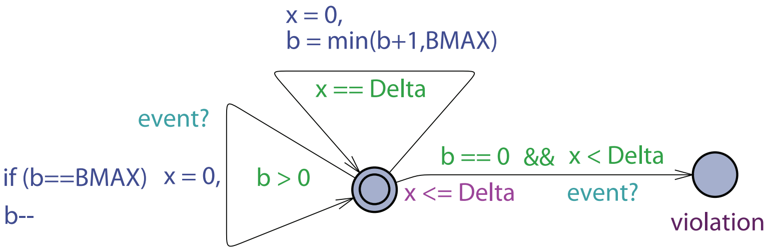
- A binary search on `estimate` (see (a) and (b)) yields the maximum burst size and the maximal pause time of the output stream, respectively.
- By fixing any of the two free parameters in the automaton of (c) or (d) and by performing a binary search on the other parameter, we obtain an upper or lower staircase bound, respectively. For example, we can use the maximum burst size from (a) in the automaton of (c) and perform a binary search on the remaining parameter δ_i^u which yields a valid upper staircase function $\alpha_i'^u$ for the output stream.

---

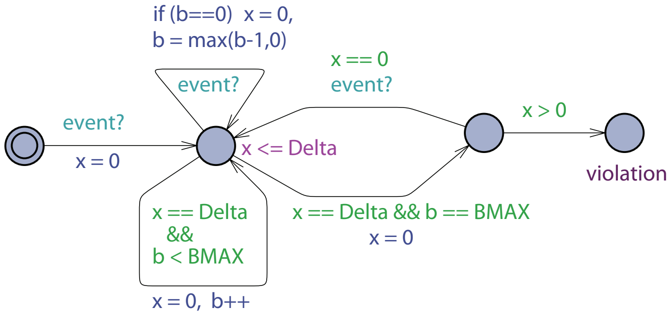
[5] In Uppaal --> stands for 'always eventually leads to'.



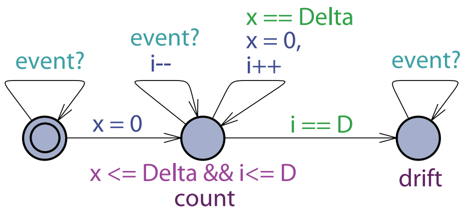
(a) Maximum burst (b) Maximum distance



(c) Upper bound



(d) Lower bound



(e) Tester for upper long-term rate

Fig. 60: Observer automata for deriving upper and lower output curves

- Another option is to use the automaton of (c) with a sufficiently large initial burst capacity N_i^u and to perform a binary search on δ_i^u. This leads to a tight upper bound on the maximal long-term rate.

- By repeatedly applying the above alternatives, one can determine a concave hull of the upper and a convex hull of the lower (unknown) arrival curves of the output event stream. For example, in case of an upper curve, we can consider a sequence of increasing values N_i^u and use the automaton (c) to determine the corresponding maximal values δ_i^u that bound the system output. The sequence ends if the long-term rate is met, see (e).

- All constructed upper and lower staircase functions can be combined to a valid arrival curve by applying minimum and maximum operations, respectively.

5.5.2 Algorithm

Let us now formalize one alternative for extracting a tuple $\alpha' = [\alpha'^u, \alpha'^l]$ of output arrival curves from a TA-based subsystem. Algorithm 1 defines a simple heuristic procedure that employs the above observer automata. We will employ this algorithm in the case study and the experiments of Section 5.6. The heuristic returns four vectors $\bar{N}^u$, $\bar{\delta}^u$, $\bar{N}^l$, $\bar{\delta}^l$ with the parameters of the linear staircase functions that bound the output event stream. The input parameters of the algorithm have the following meaning:

n, m	Maximal number of staircase functions $\alpha_i'^u$ and $\alpha_i'^l$ that shall be used to bound the output stream
B_{MIN}, B_{MAX}	Delimit the search interval for the maximum burst size
P_{MIN}, P_{MAX}	Delimit the search interval for the maximum pause between two events
$\delta_{MIN}, \delta_{MAX}$	Delimit the search interval for the parameters δ_i^l and δ_i^u
k	Scaling factor > 1 for N_i^u, N_i^l

In line 5, the heuristic determines the maximum burst N_1^u in the output event stream. This is done by means of the function `max_burst` which implements a binary search. Specifically, in line 37, the function `max_burst` calls the Uppaal model checker to verify whether in the observer TA of Figure 60(a), denoted as `OMB`, a given event counter value is never exceeded. Similarly, in line 6, the function `max_delta` determines the maximal value of δ_1^u such that the output stream never violates the bound

Algorithm 1 `Bound output of TA component` (part 1)

```
function DERIVE_BOUNDS
input: $n, m, k, B_{MIN}, B_{MAX}, P_{MIN}, P_{MAX}, \delta_{MIN}, \delta_{MAX}$
output: $\bar{N^u}, \bar{\delta^u}, \bar{N^l}, \bar{\delta^l}$
    // Upper bound
    $N^u_1$ ←max_burst($B_{MIN}, B_{MAX}$)
    $\delta^u_1$ ←max_delta($N^u_1, \delta_{MIN}, \delta_{MAX}$)
    $T^u_1 \leftarrow N^u_1$
    for $i \leftarrow 2, n$ do
        $T^u_i \leftarrow k * T^u_{i-1}$
        $\delta^u_i$ ←max_delta($T^u_i, \delta^u_{i-1}, \delta_{MAX}$)
        $N^u_i$ ←min_N_upper($\delta^u_i, N^u_{i-1}, T^u_i$)
        if isLongTermRate($\delta^u_i$) then
            break
        end if
    end for
    // Lower bound
    $P$ ←max_pause($P_{MIN}, P_{MAX}$)
    $[N^l_1, \delta^l_1]$ ←min_delta_pause($P, \delta_{MIN}, \delta_{MAX}$)
    $T^l_1 \leftarrow N^l_1$
    for $i \leftarrow 2, m$ do
        $T^l_i \leftarrow k * T^l_{i-1}$
        $\delta^l_i$ ←min_delta($T^l_i, \delta_{MIN}, \delta^l_{i-1}$)
        $N^l_i$ ←max_N_lower($\delta^l_i, N^l_{i-1}, T^l_i$)
        if isLongTermRate($\delta^l_i$) then
            break
        end if
    end for
    remove_redundant_bounds()
    return $\bar{N^u}, \bar{\delta^u}, \bar{N^l}, \bar{\delta^l}$
end function
...
```

Algorithm 1 `Bound output of TA component` (part 2)

```
...
function MAX_BURST
input: $est_{min}, est_{max}$
output: $N$
    $est \leftarrow \lceil (est_{min} + est_{max})/2 \rceil$
    repeat
        $est_{old} \leftarrow est$
        if verifyta ( A[] (OMB.count $\leq est$) ) = satisfied then
            $est_{max} \leftarrow est$
        else
            $est_{min} \leftarrow est$
        end if
        $est \leftarrow \lceil (est_{min} + est_{max})/2 \rceil$
    until $est = est_{old}$
    return $est$
end function

function MAX_DELTA
input: $N, est_{min}, est_{max}$
output: $\delta$
    $est \leftarrow \lceil (est_{min} + est_{max})/2 \rceil$
    repeat
        $est_{old} \leftarrow est$
        if verifyta ( A[] (not OUB.violation) ) = satisfied then
            $est_{min} \leftarrow est$
        else
            $est_{max} \leftarrow est$
        end if
        $est \leftarrow \lceil (est_{min} + est_{max})/2 \rceil$
    until $est = est_{old}$
    return $est_{min}$
end function
```

specified by (N_1^u, δ_1^u). Here the model checker is used to verify whether the observer TA of Figure 60(c), denoted as `OUB`, can reach its violation location (line 52). At this point, the first staircase curve $\alpha_1'^u$ is fixed. Next, the heuristic enters a loop (line 8) in which at most $n - 1$ other staircase functions are determined for the upper bound. At each loop iteration the value T_i^u is scaled by a factor k, where T_i^u represents a tentative value for N_i^u. Line 10 is equivalent to line 6. However, when looking for the largest valid δ_i^u, the algorithm considers that $\delta_i^u > \delta_{i-1}^u$. Hence, it uses tighter bounds for the binary search. In line 11 the heuristic calls the function `min_N_upper` which determines N_i^u by verifying whether the found staircase function with parameters (T_i^u, δ_i^u) can be further shifted down vertically. This may be possible, as the staircase function (T_i^u, δ_i^u) is not necessarily the smallest correct staircase function with rate δ_i^u. The function `min_N_upper` is analogous to `max_delta`, with the only difference that the binary search is carried out on N_i^u instead of δ_i^u. For conciseness, we omit the corresponding pseudo-code. After $\alpha_i'^u$ is fixed, the heuristic calls the function `isLongTermRate` which uses the TA of Figure 60(e) to verify whether δ_i^u corresponds to the long-term rate of the system output. If this is the case, it does not make sense to further increase T_i^u and the approximation of the upper bound terminates. The derivation of the lower bound follows the same line of thought, with analogous functions `max_pause`, `min_delta_pause`, and `max_N_lower` which employ the TA of Figures 60(b), 60(d), as well as an adapted version of the TA of Figure 60(e). One notable difference to the upper bound is that we cannot directly compute N_1^l given the value of P, the maximum pause between two events. In particular, there are multiple staircase functions that contain the cartesian point $(P, 0)$. Hence, in line 18, the heuristic calls the function `min_delta_pause` which looks for a curve $\alpha_1'^l$ that contains $(P, 0)$ and in addition has the smallest correct value of δ_1^l. At this point N_1^l is also determined. Finally, in line 28, the heuristic removes redundant staircase curves, that is, $\alpha_i'^u$ for which it holds

$$\exists \alpha_j'^u : ((N_j^u < N_i^u) \wedge (\delta_j^u \geq \delta_i^u)) \vee ((N_j^u \leq N_i^u) \wedge (\delta_j^u > \delta_i^u)), \tag{5.10}$$

and $\alpha_i'^l$ for which it holds

$$\exists \alpha_j'^l : ((\left|N_j^l\right| < \left|N_i^l\right|) \wedge (\delta_j^l \leq \delta_i^l)) \vee ((\left|N_j^l\right| \leq \left|N_i^l\right|) \wedge (\delta_j^u < \delta_i^u)), \tag{5.11}$$

If after termination the upper (lower) long-term rate of the system is not reached, we can either use a larger value for the parameter n (m), or try a larger value for the scaling factor k. In many practical systems, however, the long-term rates of the system output are known a priori. For instance, it is often the case that a component affects the jitter of an event stream, but not its period. In such cases, it is better to adopt an inverse search strategy in the heuristic. Specifically, for the upper bound, one would start from the known long-term rate δ_n^u and derive the corresponding value N_n^u in order to fix the last staircase function $\alpha_n'^u$. The function $\alpha_1'^u$ could be found as described before by using the maximum burst of the output stream. Successively, for refining the upper bound, one would use different values N_i^u with $N_1^u < N_i^u < N_i^u$ and derive the corresponding values δ_i^u.

5.5.3 Correctness

It remains to be shown that the heuristic of Algorithm 1 guarantees the correctness of the output interface.

Thm. 12: *Let R^{TA} be the set of event traces producible by a TA subsystem S. Let $\alpha' = [\alpha'^u, \alpha'^l]$ be a tuple of arrival curves derived for the output of S by means of Algorithm 1. Then, $R^{\alpha'} \supseteq R^{TA}$.*

Proof (Sketch). We illustrate the idea for the justification of the upper bound α'^u. The reasoning for the lower bound α'^l is analogous. Let $\bar{N}^u$ and $\bar{\delta}^u$ be the parameter vectors derived by the heuristic for the output of S. Let $\alpha_i'^u$ with $i \in \{1...n\}$ be the staircase functions defined by those parameters. It is sufficient to show that for each individual staircase curve $\alpha_i'^u$ we have $R^{\alpha_i'^u} \supseteq R^{TA}$, i.e, for each output event trace r producible by S we have $r \models \alpha_i'^u$. Consider first $\alpha_1'^u$. The function `max_burst` called in line 5 implements a binary search on the maximum burst in the output of S. By using the observer TA of Figure 60(a), the function verifies that a conservative estimate N_1^u is returned for the maximum burst in the stream. Similarly, by means of the TA of Figure 60(c), the function `max_delta` guarantees that a value δ_1^u is returned such that $\alpha_1'^u$ is never violated by the output of S. Hence, $r \models \alpha_1'^u \ \forall r \in R^{TA}$. The same argument holds also for all successive calls of `max_delta`, since the scaling factor k is such to assure $N_i^u \geq N_1^u$. Thus, $r \models \alpha_i'^u \ \forall r \in R^{TA} \ \forall i \in \{1...n\}$.

□

5.6 Experimental Evaluation

In this section we evaluate the performance of the proposed analysis methodology. We first discuss a case study that demonstrates the benefits of the hybrid analysis approach. Thereafter, we elaborate on the scalability and accuracy of the presented method.

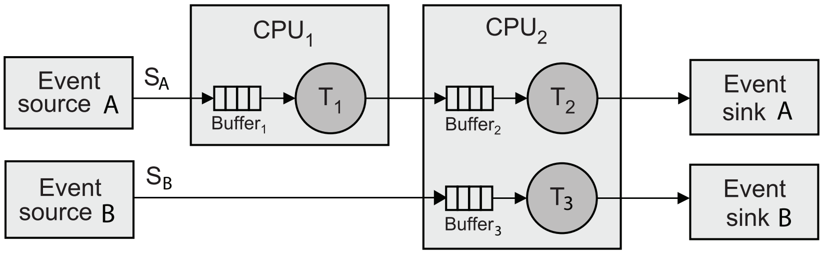


Fig. 61: System architecture

5.6.1 Case Study

The considered system is shown in Figure 61. It consists of three event-triggered tasks T_1, T_2 and T_3 that run on two distinct processors CPU_1 and CPU_2. We assume that each task is triggered by the events of the corresponding input event stream, and that it produces an event on the corresponding output event stream once its execution is completed. The three tasks process two event streams S_A and S_B which are periodic streams with large jitters that lead to bursts. S_A and S_B are specified by the parameter triples p_A = 7ms, j_A = 28ms, d_A = 1ms and p_B = 7ms, j_B = 23ms, d_B = 6ms, respectively. CPU_2 implements a preemptive fixed-priority scheduling policy, with T_2 having higher priority than T_3. The execution of each task on its respective CPU takes 10^6 cycles. CPU_2 operates at a constant frequency of 350MHz. CPU_1 implements a load-dependent frequency adaptation. Specifically, it operates at 166MHz if there are 3 events or less in its input buffer, and at 500MHz otherwise. Note that, for the sake of simplicity, we assume that the CPU frequency cannot be changed during the processing of an event. That is, the new CPU frequency is chosen only at the beginning of an event processing (depending on the current buffer fill level) and this frequency is kept constant until the next event processing starts. The goals of the performance evaluation are to characterize the event output stream of T_1, to determine the maximum delays and backlogs that events can experience at the single tasks, and to find the maximum end-to-end delay for stream S_A.

In this case study, we compare three different analysis approaches:

1. We analyze the described system with classical MPA using the RTC Toolbox.
2. We carry out the analysis with the discussed hybrid analysis approach, where we model the state-dependent behaviour of CPU_1 as TA and analyze CPU_2 with MPA.
3. We verify the performance of the entire system by means of a dedicated, holistic TA model according to the method described in

[HV06]. This method is applicable, as the input streams can be represented by means of PJD event models.

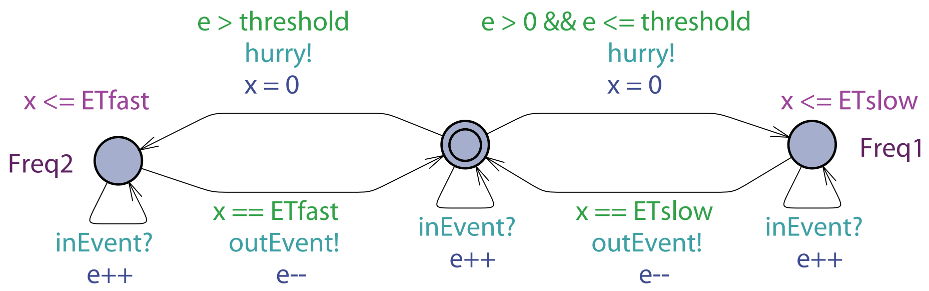


Fig. 62: TA model for CPU$_1$

For the *hybrid analysis approach* (2.), we first represent the input stream S$_A$ by the combination of three staircase functions α_1^u, α_2^u and α^l. Using the equations of Section 5.4.4, we get the parameters $N_1^u := 1$, $\delta_1^u := 1$, $N_2^u := 5$, $\delta_2^u := 7$, $N^l := -4$ and $\delta^l := 7$ for the staircase functions. The corresponding arrival curve α_{S_A} is shown in Figure 63(a). Given these parameters, we automatically create the input generator as described in Section 5.4.2. In order to increase the efficiency of the analysis, we merge the input generating network of TA into a single automaton and simplify it slightly by considering that $N_1^u = 1$, that is, for α_1^u we do actually not need a counter variable b, but just a clock to enforce a minimum distance δ_1^u between consecutive events. This input generator is then coupled with the automaton shown in Figure 62, which models the load-dependent behaviour of CPU$_1$. In this automaton, we use the signals *inEvent* and *outEvent* to distinguish between ingoing events coming from the Event Source A and outgoing events sent to T$_2$. *Buffer1* of CPU$_1$ is modelled by means of a local counter variable e. The two locations *Freq1* and *Freq2* represent the processing of events at low and high frequency, respectively, with corresponding processing times *ETslow* and *ETfast*. The signal *hurry* belongs to an urgent channel which is always ready for synchronization. This construct enforces greedy event processing. At this point, we apply the heuristic of Section 5.5.2 to get arrival curves for the output of the TA subsystem, where we choose to represent the upper bound as the minimum of three staircase functions, and the lower bound with just one staircase function. The resulting pair of arrival curves is then used as input for the MPA analysis of CPU$_2$. For the delay analysis of CPU$_1$, we customize the automaton of Figure 62 following the ideas of [HV06].

Table 6 summarizes the results of the performance evaluation. The worst-case end-to-end delay of stream S_A is denoted as EE$_A$. Note that in general for a sequence of components, the worst-case end-to-end delay

	Max delay [ms]				Max buffer [events]		
	T_1	T_2	T_3	EE_A	T_1	T_2	T_3
MPA	29	8	28.6	31.9	5	3	5
Hybrid	25	5.5	17.2	30.5	5	2	3
TA	25	4.6	14.3	27.9	5	2	3

Tab. 6: Results of Performance Evaluation

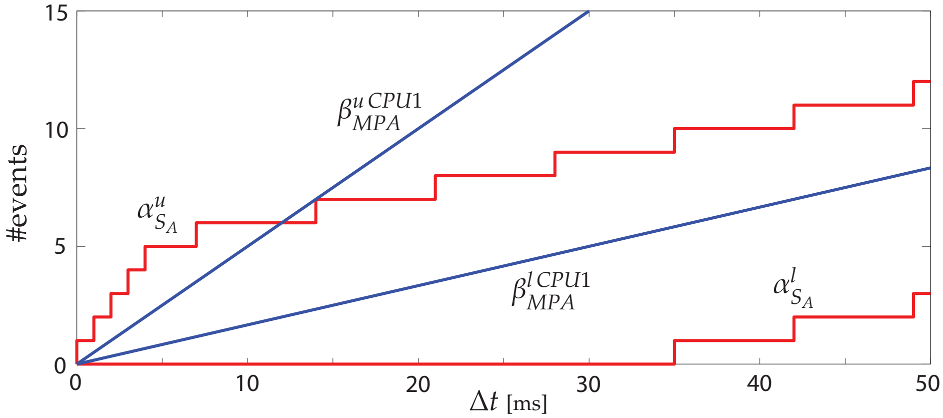
can be smaller than the sum of the individual worst-case delays (see Section 2.2.3.6). While in the abstractions of MPA and TA this phenomenon can be captured, this is obviously not possible in the hybrid approach. The table shows that in terms of accuracy the *hybrid approach* is clearly better than the pure MPA analysis. In particular, the conservativeness of the results is highly reduced, with a maximum delay and backlog at T_2 that are 31% and 33% lower with respect to the MPA analysis, respectively. For the delay and the backlog at T_3, the hybrid approach achieves values that are 40% lower compared to the pure MPA analysis.

Let us explain the reason for the better results with the help of Figure 63. A *pure MPA-based analysis* of the system cannot capture the load-dependent behaviour of CPU_1. Hence, one has to assume that the processor always operates at 500MHz in the best case, and at 166MHz in the worst case. This assumption corresponds to using the service curves β^{uCPU1}_{MPA} and β^{lCPU1}_{MPA} (cf. Figure 63(a)) for the analysis of CPU_1. This yields conservative worst-case processing load predictions for T_2, captured by $\alpha'^{u}_{1,MPA}$. However, a TA-based analysis of CPU_1 produces tighter input bounds for T_2, captured by $\alpha'^{u}_{1,Hybrid}$. This leads to smaller worst case delay guarantees in the hybrid analysis, as shown in Figures 63(b) and 63(c).

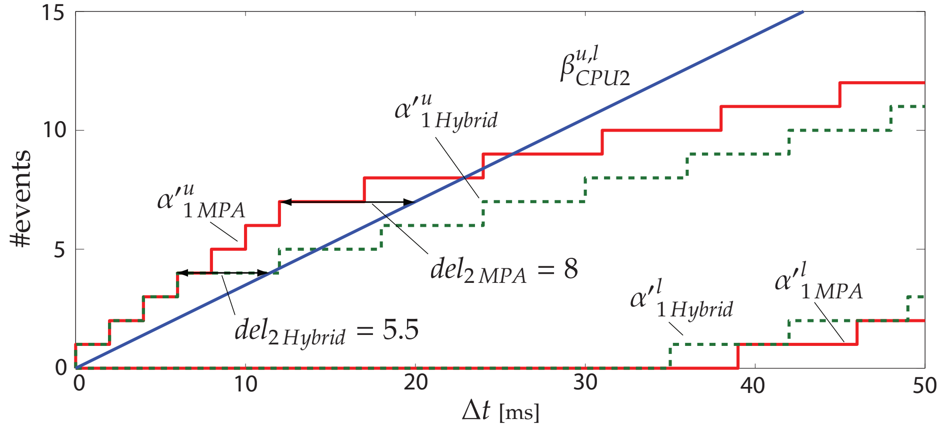
The last line of Table 6 contains the exact values for the worst-case performance of the system. These values are determined by means of the *dedicated TA model for the entire system*. As can be seen in the table, the results of the hybrid analysis are slightly more conservative. The reason is that the concave (convex) hull determined as bound for the output event stream of T_1 does slightly over- (under)-approximate the real behaviour of the system.

The higher degree of accuracy of the hybrid analysis method has its price, namely a substantially longer run-time compared to pure MPA, see Table 7. This becomes worse if one requires an even higher accuracy for the hybrid analysis, e.g. more staircase functions or non-concave/convex patterns for the arrival curves. Nevertheless, the run-times achieved for the hybrid approach are still significantly better compared to the verification of the pure TA model.

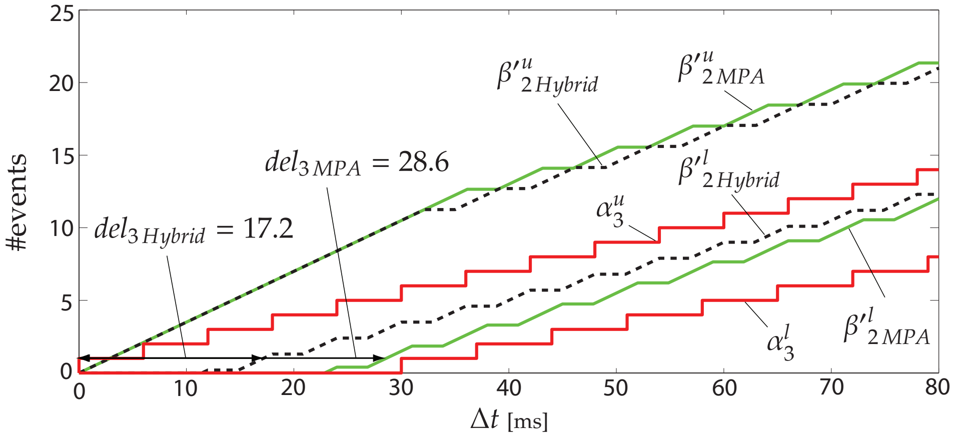
Furthermore, we found that in the hybrid approach, the run-times



(a) Arrival and (conservative) service curve for T_1



(b) Delay computation for T_2



(c) Delay computation for T_3

Fig. 63: Curves associated with the case study

	MPA	Hybrid	TA
Total run-time	< 1s	11min	1h

Tab. 7: Run-times for Performance Evaluation (referred to a commodity PC with a dual core CPU and 2GB of RAM)

to derive an output curve from a TA component can be considerably reduced if, for the representation of the input stream, we omit the lower bound. In other words, we can speed up the procedure by employing only UTA in the input event generator, and leaving out LTA. In this case study, this corresponds to representing the event stream S_A by the upper bound $\alpha^u_{S_A}$ of Figure 63(a) only, without specifying the lower bound $\alpha^l_{S_A}$. Such a relaxation of the stream specification does not harm the correctness of the analysis. In particular, by omitting the lower bound, we specify a superset of input traces with respect to the case with both upper and lower bounds. In other words, all behaviours of the original model are contained in the relaxed model, and hence the analysis is safe. However, depending on the behaviour of the modelled system component, considering more input streams than in the original model might lead to more conservative analysis results. In the system of Figure 61 this is not the case, meaning that the same analysis results are achieved when representing the stream S_A with UTA only. In terms of verification effort the difference is, however, substantial; by leaving out the LTA, the run-time of the hybrid approach is reduced from 11 min to 18 s. This shows that the synchronization of UTA and LTA in the input generator is the major source of complexity for the discussed verification method.

5.6.2 Scalability of the Approach

In the following, we report the results of two different experiments that investigate the scalability of the proposed analysis method. The first experiment demonstrates that the presented compositional methodology is clearly superior to holistic TA models in terms of scalability of the verification effort. The second experiment points out a main limitation of TA-based performance evaluation in general, namely the blow-up of the verification effort with increasing non-determinism in the system specification.

5.6.2.1 Modular vs. Holistic TA verification

In this experiment we consider a larger distributed system consisting of several state-based components. We compare two different TA-based methods for the analysis of the system. The first approach performs a holistic analysis based on a single TA model of the entire system. In the second approach, the analysis is strictly modular. In particular, in the second case each component of the system is analyzed separately by an individual TA model, where we use the described interfaces based on staircase-functions to represent the input and output event streams of the components. Obviously, it comports some verification overhead to explicitly characterize the input/output interfaces of each component by appropriate staircase functions. However, we still expect better scalability for the modular approach, as in contrast to the holistic method, the analysis of a component is totally decoupled from other components. In order to highlight how well the two different approaches scale with the size of systems, we gradually increase the number of components in a predefined system architecture, and compare the results and run-times of the analysis methods.

The considered system template is a chain of n tasks, where each task executes on a dedicated processor. We assume that the execution of each task takes 10^6 processor cycles. The tasks are arranged one after another and process the events of an input event stream S. Figure 64 shows an instance of the system for $n = 5$.

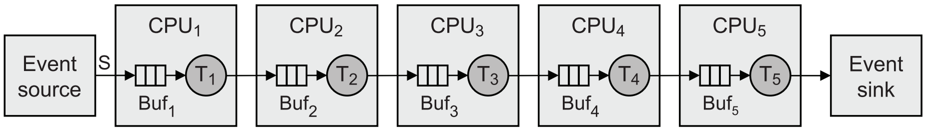


Fig. 64: System instance with five components

Each CPU in the chain implements a load-dependent frequency adaptation (see details below). For the experiments, we consider five different system instances, from $n = 1$ to $n = 5$. That is, the first instance consists of T1/CPU1 only, the second instance of T1/CPU1 and T2/CPU2 etc. In order to allow for event bursts also at the last components of the chain, we choose different maximum frequencies for the five processors. The parameters for the processors are summed up in Table 8. The load-dependent frequency adaptation works as follows: If there are not more than *threshold* events in the input buffer of a CPU, the CPU executes at frequency f_{low}, otherwise at f_{high}. We again exclude frequency changes during the processing of an event.

The aim of the performance evaluation is to determine the worst-case backlogs at the single event buffers for each system instance. The considered system has a pure feed-forward architecture. Hence, when we extend it by adding one component at the end of the chain, we need to verify only the backlog of the new component. This is because previous components are not affected by the extension of the system.

	CPU_1	CPU_2	CPU_3	CPU_4	CPU_5
f_{low} [MHz]	166	166	166	166	166
f_{high} [MHz]	1000	500	333	1000	500
threshold [events]	1	1	1	1	1

Tab. 8: Parameters for the CPU chain

For the input event stream S, we assume the same upper bound as for S_A in the previous case study, that is, $\alpha_S^u = \alpha_{S_A}^u$. In order to speed up the run-times for both the holistic and the modular analysis, we do, however, omit the specification of the lower bound α_S^l (see comment at the end of Section 5.6.1). The resulting TA model for the input generator consists of two UTAs with parameters $N_1^u = 1$, $\delta_1^u = 1$, and $N_2^u = 5$, $\delta_2^u = 7$.

	Buf_1	Buf_2	Buf_3	Buf_4	Buf_5
TA holistic	5	5	4	4	3
TA modular	5	5	5	4	5
MPA	5	6	6	6	7

Tab. 9: Worst-case backlogs as derived with the different approaches

The results of the performance evaluation are reported in Table 9. The first row in the table contains the exact values for the worst-case backlogs. These values are determined by means of holistic TA models for the different system instances. The second row shows the worst-case backlogs as predicted by the modular analysis approach based on TA. The reason for the slightly more conservative results is the same as in the case study of Section 5.6.1: The concave hulls derived as upper bounds for the event streams transmitted between components are an over-approximation of the real streams. In particular, for the sake of efficiency, we decided to represent each input/output stream with a concave pattern of two linear staircase functions only. This is not sufficient to capture the exact behaviour of the streams. For comparison, in the last row of Table 9 we also report the analysis results achieved by an MPA analysis of the system instances. This analysis is obviously penalized, as the state-based behaviour of the components cannot be captured in the MPA models.

Let us now focus on the computational effort required by the considered analysis approaches. Figure 65 displays the run-times of the different methods for the analysis of the five system instances. These run-times are cumulative, meaning that for a system instance with n components they express the total time needed to determine the worst-case backlog values for all n buffers. For the holistic TA analysis we consider two different alternatives for the modelling of the input generator. The first variant uses the staircase-based TA pattern for event generation described before, which in this case corresponds to the combination of two UTA. The second variant uses an optimized input generator for periodic event streams with jitter/bursts as described in [HV06]. The chart of Figure 65 shows a clear trend for the holistic analysis approaches: The run-times increase exponentially with the size of the considered system instance (note the logarithmic scale on the y-axis). This holds for both types of input event generators, the general one based on UTA, and the optimized one designed for PJD streams. When we use the general input generator to trigger the holistic TA model, we report a run-time of more than two hours for analyzing the first three components. For system instances with more than three components, the model checker runs out of memory after several hours of verification. For the optimized input generator the run-times are slightly better with a maximum bearable system size of five components.

Also for the modular TA-based analysis approach, we can identify a trend: The run-times increase nearly linearly with the number of considered components. In particular, for each additional component in the chain, the run-time increases by roughly 4-30 s. Given the concave hull that describes the input stream of a component, this is the time needed to determine the worst-case backlog of the component, and to derive the concave hull that bounds the output stream. The deviations from an exact linear increase are supposedly a consequence of the varying amount of non-determinism present in the input streams at the different stages.

The above experiment highlights one of the main advantages of the proposed analysis framework: It enables a fully compositional system analysis by adopting appropriate patterns to represent the input/output interfaces of components. As a result, the state-space explosion is limited to the level of isolated components. Consequently, the proposed analysis technique scales to systems of almost arbitrary size, provided that the TA abstractions of the single components are reasonably simple and that the representation of the event streams is reasonably coarse.

5.6.2.2 Non-determinism in event stream specifications

In this second experiment we investigate how sensitive the run-times of the proposed compositional analysis method are with respect to increasing non-determinism in the specification of the input event streams. In order to do so, we gradually increase the burstiness of the input event stream for a simple TA component, and measure the run-time needed to characterize the corresponding output stream.

We consider the component T1/CPU1 from Figure 61 that implements the load-dependent frequency adaptation described in Section 5.6.1. As

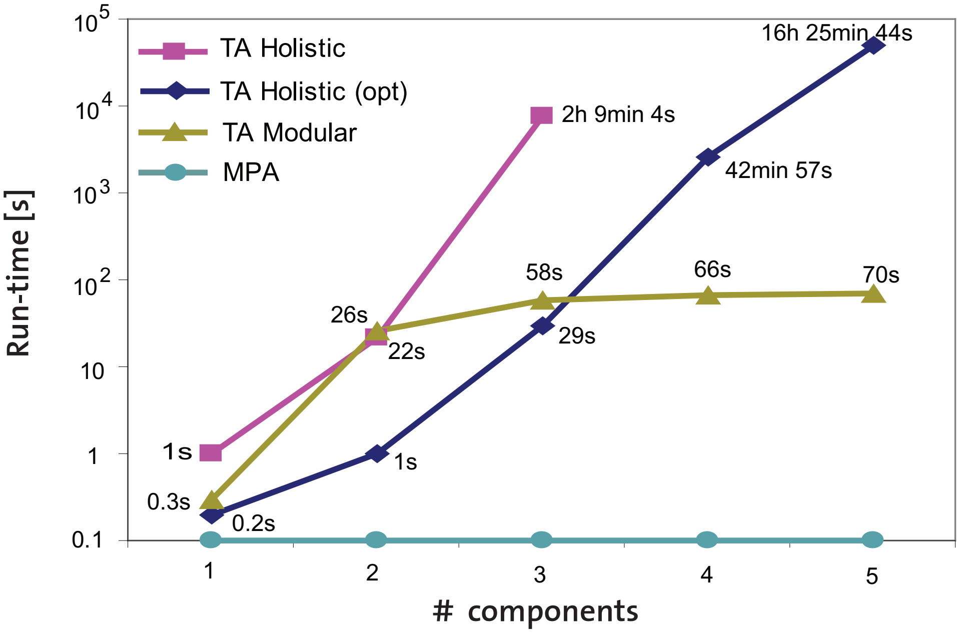


Fig. 65: Computational effort of the modular and the holistic approaches

input to the component, we consider event streams upper bounded by a simple linear staircase function with step-width $\delta^u = 7$, and seven different levels of burstiness varying from $N^u = 5$ to $N^u = 150$. As for the previous experiment, in order to speed up the verification times, we consider only an upper bound for the input event stream, and omit the lower bound. For all seven different input bounds we record the run-time needed by the heuristic described in Section 5.5.2 to characterize the output event stream, where we choose to represent the output bounds as the minimum of two linear staircase functions. In order to ensure that in all seven cases the same number of verification steps is needed to characterize the system output, we set $k := N^u$ in the heuristic.

The results of the experiment are shown in Figure 66. As can be seen in the chart, the total run-time needed to characterize the output stream increases exponentially with the jitter/burstiness of the input stream. For the input stream with $N^u = 5$, the derivation of the output event stream is performed in roughly one second. For the input stream with $N^u = 125$, we record a run-time that is three orders of magnitude larger. For the input stream with $N^u = 150$, the model checker runs out of memory.

The described experiment clearly shows a limitation of TA-based performance evaluation: Only event streams with mediocre degree of non-determinism regarding the timing of event arrivals can be handled with reasonable verification effort. This result is not very surprising, as with increasing non-determinism in a TA model, the model checker has to

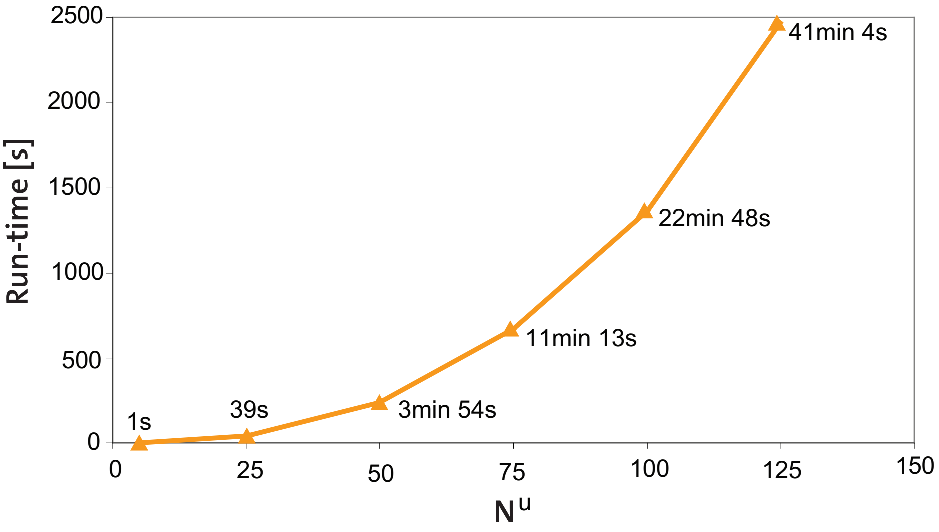


Fig. 66: Total run-time needed to characterize α

explore a larger number of system states.

5.6.3 Approximation Errors

In this final part of the experimental evaluation of the proposed method, we briefly elaborate on possible approximation errors introduced by bounding the output streams of system components with a convex/concave hull of staircase functions as described in Section 5.5. In order to characterize these approximation errors in isolation from other effects, we apply the described TA-based analysis approach to two systems consisting of stateless components only. We compare the obtained bounds with the results of an MPA analysis, which for the considered systems ensures tight results.

Consider first the simple system architecture shown in Figure 67. The depicted system consists of a CPU that executes two tasks T_1 and T_2. The two tasks are triggered by two strictly periodic streams S_1 and S_2 with periods $p_1 = 60\text{ms}$ and $p_2 = 5\text{ms}$, respectively. The CPU schedules the two tasks according to a preemptive fixed priority scheme, where T_1 has higher priority than T_2. We assume that the CPU executes at a constant frequency of 1GHz, and that the execution of T_1 and T_2 takes $60 \cdot 10^6$ and $5 \cdot 10^6$ cycles, respectively. The goal of the analysis is to characterize the output event stream S_2'. For the TA-based analysis of the system we employ a holistic TA model for the preemptive fixed priority scheduling of two tasks, as described in [Per06].

Figure 68 shows the result for both the MPA analysis and the TA

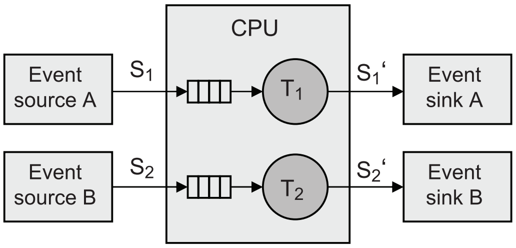


Fig. 67: Fixed priority scheduling of two tasks

heuristic of Section 5.5.2 . The curves $[\alpha'^{l}_{2,\,\mathrm{MPA}}, \alpha'^{u}_{2,\,\mathrm{MPA}}]$ (depicted with a solid line in the plot) represent the exact lower and upper arrival curves for the stream S'_2 computed by the MPA analysis. The dashed lines in the plot represent the bounds for the output event stream derived by the heuristic, where we decided to represent the upper bound $\alpha'^{u}_{2\,\mathrm{TA}}$ as the minimum of two linear staircase functions and the lower bound $\alpha'^{l}_{2\,\mathrm{TA}}$ by one single linear staircase function. As can be seen in the plot, the heuristic clearly over-approximates the real upper bound for S'_2. The reason for this abstraction loss (represented by grey shaded areas in the figure) is that the heuristic constructs only a concave hull of linear staircase functions to upper bound the output stream, whereas the real upper bound of the stream does not have a strictly concave shape. To avoid such losses, we could think of extending the heuristic of Section 5.5.2 such that it handles mixed convex/concave output patterns. However, such an extension is not trivial and would obviously also slow down the verification process considerably.

Let us now perform a second experiment to illustrate a different kind of approximation error. Consider a simplified version of the component T1/CPU1 from Figure 61. Assume that instead of the described load-dependent frequency adaptation, CPU1 can arbitrarily change its execution frequency between 166MHz and 500MHz. Such a stateless best-case/worst-case component description is ideally suited to an exact MPA analysis of the component. As input for the component we consider the stream S_A as given in Section 5.6.1, that is, a periodic event stream with jitter specified by the parameter triple p = 7ms, j = 28ms, d = 1ms. The goal of the analysis is again to characterize the output event stream of the component. Figure 69 shows the results of both approaches, the MPA analysis and the TA heuristic of Section 5.5.2. The curves $[\alpha'^{l}_{\mathrm{MPA}}, \alpha'^{u}_{\mathrm{MPA}}]$ represent the exact lower and upper arrival curves for the component output computed by MPA. These curves correspond to a periodic event stream with jitter specified by the parameter triple p′ = 7ms, j′ = 32ms, d′ = 2ms. For the heuristic approach, we decide to represent the upper bound $\alpha'^{u}_{\mathrm{TA}}$ as the minimum of two linear staircase functions and the

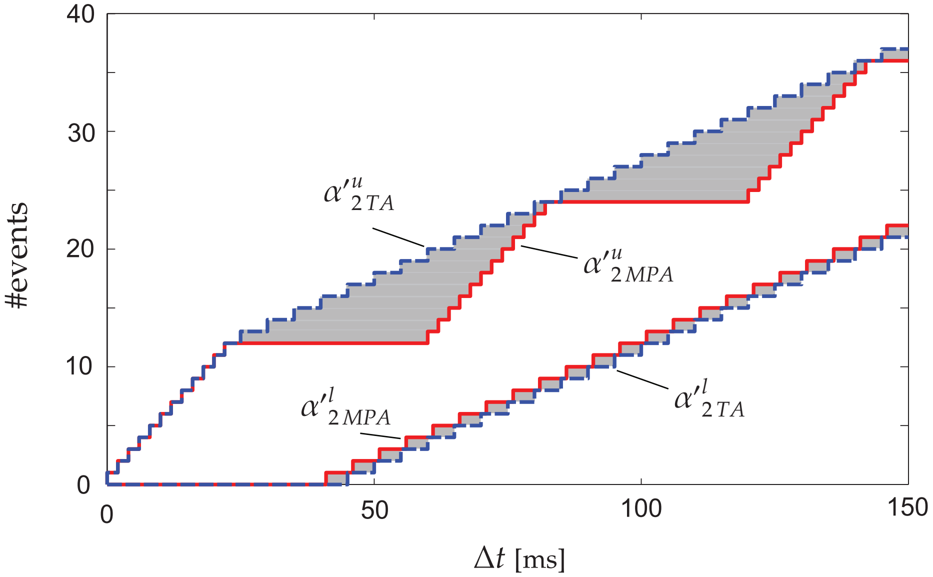


Fig. 68: Bounds for S'_2 determined by MPA (exact) and the TA-heuristic

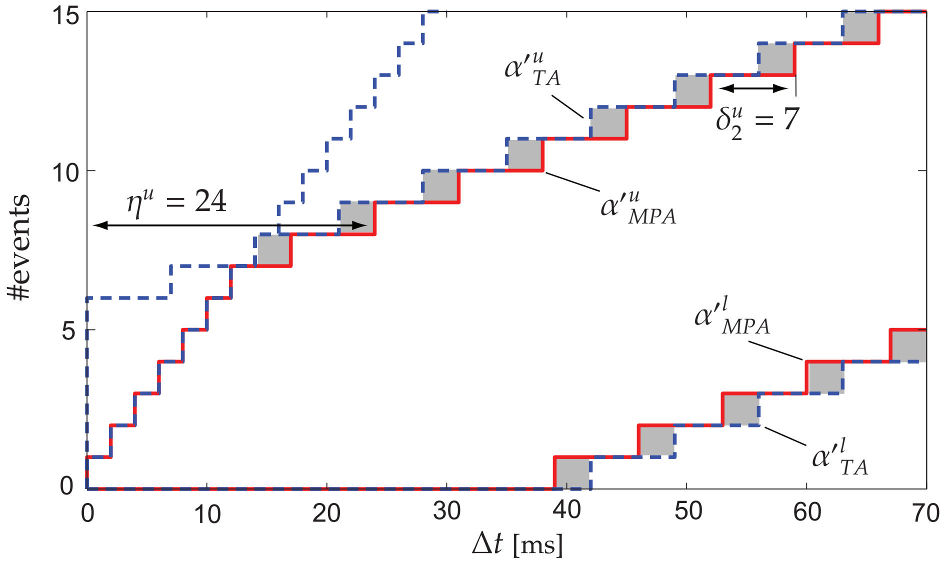


Fig. 69: Output bounds determined by MPA (exact) and the TA-heuristic

lower bound ${\alpha'}^{l}_{\mathrm{TA}}$ by one linear staircase function. The plot shows that the heuristic slightly over-approximates the real upper bound, although the maximum component output follows a concave pattern. Similarly, the lower bound is slightly under-approximated. The reason for this abstraction loss is that the heuristic of Section 5.5.2 does not consider horizontal translations of linear staircase functions. In particular, looking at Figure 69, we see that the offset η^u, after which the real upper bound of the component output follows the long-term rate δ^u_2, is not a multiple of the long-term rate itself. In fact, without horizontal offset no linear staircase function α'^u can precisely capture the long-term behaviour of the component. The reason for the under-approximation of the lower bound is analogous.

In Section 5.4.3.1 we have described how this kind of approximation error can be avoided when converting known *input* event streams to TA input generators. The case of bounding the *output* stream of a TA component is, however, more difficult, as the stream that needs to be bounded is obviously not known a priori. In particular, permitting arbitrary horizontal shifts for the linear output staircase functions would mean adding another degree of freedom for the search heuristic. This would also clearly slow down the analysis process.

5.7 Related work

There are a few other methods that tackle the combination of analytical performance evaluation and state-based system verification. The authors of [PCTT07] present an approach that permits them to convert arrival curves as used in MPA and Network Calculus to Event Count Automata (ECA) [CPT05], a finite state representation of event streams. An ECA is an ordinary FSM, augmented with counter variables. It specifies the minimum and maximum number of events that arrive in a stream, while the automaton is in a given location. The ECA takes transitions at discrete time points, based on the current values of its counters. For converting an arrival curve to an ECA, the authors of [PCTT07] propose a procedure that relies on the finite data type for arrival curves adopted in the RTC Toolbox (see Section 2.2.3.7). The procedure automatically transforms successive vertices of the arrival curve to a sequence of ECA locations. The opposite conversion, i.e., the transformation of an ECA to an arrival curve, relies on dedicated observer ECA and binary search. By means of ordinary reachability analysis, the procedure extracts the minimum and maximum number of events that may arrive in an interval of a particular length. In [PCT08] it is shown how this approach can be applied to the analysis of multi-mode real-time systems. Compared to the ECA formalism, our usage of TA appears more advantageous for the modelling of real-time systems. The reason is that TA have an explicit notion of time, whereas ECA advance in a lock-step fashion. Moreover, the output interface of our method requires only one observer automaton for an entire linear staircase function, whereas with ECA one observer is needed for each discrete interval size.

In [DMS09] another approach is presented that exploits the advantages of both analytical performance evaluation and state-based system verification. The method is not hybrid at the component level. Rather, it applies the different formalisms at different stages of the system evaluation. Specifically, it first maps the system under evaluation to a process network which is analyzed via compositional response time analysis [HHJ$^+$05]. The resulting event models and response times are then used to parameterize a pre-defined TA model of the system automatically. At this point the system model is coupled with other TA models that express the system properties to be verified. Finally, ordinary model checking is used to ascertain the correctness of the system. A major drawback of this approach is that it cannot explicitly represent components with state-based behaviour. Rather, it uses TA models as intermediate representation for intrinsically stateless systems, which makes the application of TA questionable.

The authors of [KMY07] also address the combination of MPA and TA. For encoding arrival curves by means of TA, they use a circular array of clocks as basic data structure. These clocks keep track of the time that passed since the generation of the last events. The bounds imposed by the arrival curve are encoded by appropriate constraints on the clock variables. By observing these constraints, the input generator can produce only event traces that conform to the arrival curve. The main drawback of this kind of input generators is that they may require a prohibitive number of clocks. In fact, one needs one clock for each vertex of the modelled arrive curve. To extract arrival curves from a TA model, the authors of [KMY07] suggest the use of observer automata and binary search. Specifically, they employ model checking to verify the minimum/maximum number of events that can appear on the output stream in an interval of a particular size. This output interface is basically equivalent to the one of [PCTT07]. Its major drawback is that it can characterize the system output for a finite set of interval sizes only, as it verifies the output arrival curve vertex by vertex. In this chapter, we have overcome these limitations by decomposing arrival curves to simpler curve components. The decomposition permits us to employ single clock variables for representing entire curve components, and not just single vertices of a curve. As has been demonstrated, this leads to considerably more scalable system models.

5.8 Summary

In this chapter we introduced a hybrid analysis method that combines analytic performance evaluation with state-based system verification. In the resulting modular framework, system components are abstracted either analytically by means of MPA or with state-based models in the form of Timed Automata. The proposed methodology permits us to considerably reduce the abstraction losses experienced with MPA due to coarse abstractions of components with state-dependent behaviour. At the same time, it limits state space explosion, as intrinsic to formal verification, to the level of individual system components. To maintain the scalability of the approach, we suggest employing detailed TA models exclusively for those components where an MPA analysis is too pessimistic. In short, our method can balance the accuracy and the complexity of the performance evaluation.

The presented technique derives from the observation that arrival curves can be represented as sets of linear staircase functions which are composed by minimum and maximum operations. The method is based on two interfaces, the input interface MPA→TA and the output interface TA→MPA, that were extensively discussed. The input interface converts an arrival curve to an input generator, that is, to a network of TA that generates event traces and that is used to trigger a user-defined TA component model. In the input generator, each linear staircase component of the arrival curve is guarded by a dedicated TA. To represent the minimum and maximum of staircase functions, appropriate synchronization mechanisms are employed. The output interface performs the inverse transformation, that is, it constructs valid arrival curves for the output of a TA component. This is done by employing appropriate observe automata in a binary search based heuristic. For both interfaces we have proven correctness, which means that the hybrid framework delivers hard performance guarantees for the modelled systems. As the class of arrival curves includes other event stream models such as the widely used PJD (periodic with jitter) model, the proposed method can also be directly applied for coupling TA-based system verification with MAST or SymTA/S.

By means of a simple case study, we demonstrated that our method achieves more accurate performance bounds for systems with state-dependent components, while still being more efficient than the verification of a holistic TA model. We also performed experiments to investigate the scalability of the proposed method, and to identify potential inaccuracies of the conversions.

Finally, let us indicate some issues that our work leaves open. The heuristic devised for the output interface does not explore shifted staircase curves or non-convex/concave patterns. In more general terms, the present work does not assure the tightness of the output interface. It does also not consider cycles in the event flow or non-functional cyclic dependencies among components as considered in Chapter 3. These matters are left for future work.

6

Energy-Efficient System Design with MPA

So far, our discussion of performance evaluation has concerned only the timing properties and memory requirements of embedded systems. However, as mentioned in the introduction, there are also other performance requirements that many systems need to meet. One fundamental aspect for several embedded systems is their energy consumption. For instance, the amount of dissipated power directly affects the battery lifetime of mobile devices such as cell phones or PDAs. During the last decade, many techniques have been proposed to reduce the energy consumption of real-time embedded systems. Examples are the real-time adaptations of Dynamic Voltage Scaling (DVS) and Dynamic Power Management (DPM). However, most of these techniques rely on stringent assumptions such as deterministic input event streams. In this chapter, we consider the design and analysis of energy-efficient real-time systems which process *non-deterministic* input streams. We propose novel energy-aware design methods, which are based on MPA and TA. We focus on DVS, but similar results may be obtained for DPM. Specifically, we consider two distinct design problems: (1) The energy-efficient *offline* assignment of execution speeds and priorities to a set of real-time tasks triggered by event streams; (2) The energy-efficient *online* adjustment of the execution speed of a task that processes an event stream. The first method is based on the MPA framework discussed in Chapter 2. The second one builds on top of the hybrid evaluation framework introduced in Chapter 5.

6.1 Introduction

Power dissipation increasingly limits the performance of modern computing systems. Therefore, power management in both hardware and software has become one of the primary aspects of system design. Effective power management can significantly prolong the battery lifetime of autonomous embedded systems or reduce the power bill of server systems. The dynamic energy consumption of a system can be reduced by means of dynamic voltage scaling (DVS), a technique that balances system performance and energy savings. In general, a lower supply voltage for a processor leads not only to a lower execution speed, but also to a lower power consumption. Hence, most DVS scheduling algorithms, e.g. [YDS95], tend to execute events/jobs as slowly as possible in order to save energy. On the other hand, to consume less leakage (static) power, one can apply dynamic power management (DPM). In DPM systems the processor can switch among different power modes such as running, idle or sleep. DPM algorithms, e.g. [JPG04], tend to aggregate jobs and typically put the system into a sleep mode when there are no events/jobs to process. In our work we focus on DVS in real-time systems.

For real-time systems, energy-efficient design stands for minimizing the energy consumption without violating the deadlines of tasks. Known techniques can broadly be divided into the two main classes of offline and online methods. Offline approaches determine the system behaviour statically by assuming worst-case workload for tasks, see, e.g., [YDS95]. On the other hand, online approaches reclaim energy dynamically by adapting to the actual workload of a system, see, e.g., [AMMMA01].

Most studies for energy-efficient scheduling in real-time systems assume that the input events arrive according to regular patterns, i.e., periodically or sporadically. Another common assumption is to consider irregular arrival patterns, but with full a priori knowledge of event arrival times. Unfortunately, both assumptions are not realistic for many systems. In fact, event arrivals in practical systems are often neither regular nor fully predictable. There are two main reasons for non-deterministic event arrival times:

(a) Tasks of embedded systems are often triggered by the physical environment which can, in general, not be predicted accurately

(b) Variable execution demands, communication delays, and interference on shared resources all make it extremely difficult to predict precise activation times of individual tasks in distributed architectures.

In the domain of performance analysis of distributed embedded systems, powerful abstractions have been developed for capturing the timing non-determinism of event streams. Examples are the PJD event model and the abstraction of arrival curves described in Chapter 2.

In this chapter we propose novel design methods for energy-efficient real-time systems with non-deterministic event streams. Our methods employ the concept of arrival curves to represent the bounded non-determinism of event streams. The proposed techniques target the efficient DVS scheduling of event streams and rely either on MPA or TA to guarantee the observation of timing constraints. Since arrival curves permit the characterization of *arbitrary* event streams, our methods considerably extend the modelling scope of existing DVS techniques. In particular, we look at the following two design problems:

1. ***Offline DVS scheduling***
   Given is a monoprocessor system with multiple non-deterministic input event streams and real-time requirements. The processor schedules the streams with static priorities and preemption. Assign static execution speeds and priorities to the streams such that the timing and speed constraints are met, and the energy consumption is minimized.

2. ***Online DVS scheduling***
   Given is a monoprocessor system that executes a single non-deterministic input event stream with real-time requirements. Adapt the execution speed dynamically such that the timing and speed constraints are met, and the energy consumption is minimized.

6.1.1 Organization

The contents of this chapter are organized as follows. In Section 6.2 we give a brief introduction to Dynamic Voltage Scaling. We also describe two existing approaches for offline and online DVS scheduling of single event streams, which form the basis of our contributions. In Section 6.3, we tackle design problem 1. Specifically, we propose different heuristics for the design of systems with offline DVS and static priorities. In Section 6.4, we approach design problem 2. We propose an adaptive scheme that combines pessimistic and optimistic DVS scheduling to execute a real-time event stream. The scheme ensures that both timing and speed constrains are met and at the same time ensures an energy-efficient execution. Finally, we discuss related work in Section 6.5 and provide a summary of the chapter in Section 6.6.

6.2 Dynamic Voltage Scaling

The switching activity in CMOS circuits leads to dynamic power dissipation due to charging and discharging of load capacitances. Specifically, for the dynamic power dissipation P_d of a CMOS processor it holds [CSB92]

$$P_d \sim a\, C_L\, V_{dd}^2\, f, \tag{6.1}$$

where V_{dd} is the supply voltage, a is the switching activity, C_L is the load capacitance, and f is the clock frequency of the processor. Note that the above term does not consider any static components of the power dissipation, such as leakage currents. Consequently, the dynamic energy consumption for the execution of a task with processing time t (at frequency f) fulfills

$$E_d \sim a\, C_L\, V_{dd}^2\, f\, t = a\, C_L\, V_{dd}^2\, n, \tag{6.2}$$

where n corresponds to the number of processing cycles required for the execution of the task. On the other hand, for the delay d of CMOS circuits we have

$$d = k\, C_L\, \frac{V_{dd}}{(V_{dd} - V_T)^2}, \tag{6.3}$$

where k is a positive constant and V_T is the threshold voltage with $V_T \ll V_{dd}$. From Equations 6.2 and 6.3, it follows that by decreasing the supply voltage V_{dd} one can reduce the dynamic energy consumption E_d *quadratically*, whereas the gate delay d will increase only *linearly*. This fact is exploited by the technique of Dynamic Voltage Scaling (DVS), which achieves energy savings by reducing the supply voltage of the processor. A linear increase of the circuit delay translates to a linear decrease of the maximal practicable processor frequency f_{max}, and hence also to a linear increase of the processing time of a task. In short, one can save energy by executing tasks at a slower pace. In real-time systems, the adaptation of the execution pace must obviously be such that the timing constraints are met. In the reminder of this chapter we express the execution pace of a CPU not by its clock frequency f, but by a speed s that denotes the amount of workload units that the CPU processes per time unit.

In the following, we summarize two existing DVS approaches for executing a single non-deterministic event stream with real-time constraints. The first approach determines the execution speed offline (at design time) whereas the second one adopts the execution speed online (at run time). These methods form the basis for our contributions in Sections 6.3 and 6.4.

6.2.1 Offline DVS scheduling

The offline DVS scheduling of a single non-deterministic event stream is considered in [MCT05]. The method models the stream by means of an arrival curve and employs MPA to derive the minimum speed that guarantees the deadlines. Since from the arrival curve one cannot infer the actual arrival times of individual events in the stream, the method determines the minimum *constant* execution speed that ensures the schedulability of all possible event traces.

The minimum constant execution speed is found by considering the expression for the worst-case response time (or maximum delay) of a GPC component $Del(\alpha, \beta)$ (cf. Equation (2.41)), where α is the upper arrival curve of the non-deterministic event stream and β is the lower service curve of the processor[1]. The stream is *schedulable* if $Del(\alpha, \beta) \leq D$. In other words, all timing constraints are met if the worst-case response time for the stream is not larger than its relative deadline D. Remember that $Del(\alpha, \beta)$ corresponds to the maximal horizontal distance between α and β. Hence, the schedulability condition can also be expressed as

$$\alpha(\Delta - D) \leq \beta(\Delta) \quad \forall \Delta \geq 0, \tag{6.4}$$

where $\alpha(\Delta - D)$ is the upper arrival curve translated by the deadline D, see Figure 70. Based on the above analysis, the offline DVS schedule consists in executing the stream at a constant speed s_{SD} such that

$$\alpha(\Delta - D) \leq s_{SD} \cdot \Delta \quad \forall \Delta \geq 0. \tag{6.5}$$

Figure 70 illustrates an example of the above strategy. Executing the event stream at the constant speed s_{SD} ensures that all deadlines are met, even under worst-case arrivals of events. In the following we denote this offline approach as *Algorithm SD* which stands for static DVS.

Note that a concrete event trace of the non-deterministic stream might be far below the arrival curve α in many time intervals. This is because α bounds the worst-case event trace. In such a case, Algorithm SD still executes the trace at constant speed s_{SD}, that is, faster than actually necessary. In other words, by means of a static speed assignment one tends to be too pessimistic. For this reason, we call the above offline approach also *pessimistic DVS scheduling*.

6.2.2 Online DVS scheduling

In contrast to an offline scheme, an online DVS scheduling algorithm takes a new scheduling decision each time an event *really* arrives. Hence, an

---

[1] In this chapter we consider only upper arrival curves and lower service curves. We will therefore omit the superscripts u and l.

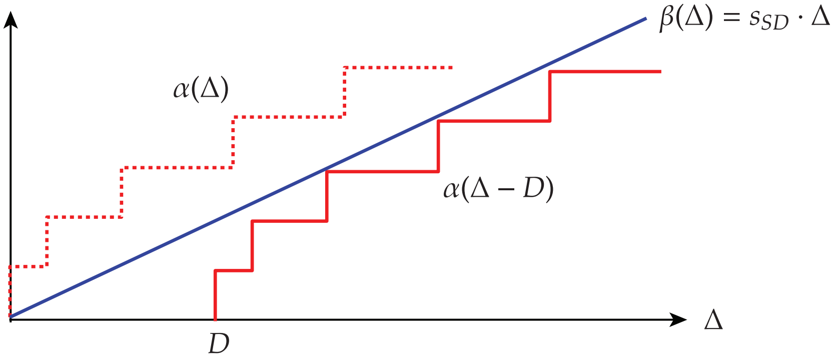


Fig. 70: Graphical illustration for the derivation of s_{SD}

online DVS algorithm can reduce the energy consumption of a system by adapting the CPU speed to the actual trace of input events. Different online DVS algorithms have been proposed in the literature [YDS95, BP05]. A well-known online DVS scheme is the *Algorithm OPT* proposed by Yao et al. [YDS95]. In the following, we briefly summarize the behaviour of this algorithm.

For an event e_j that is not yet fully processed at a time t, suppose that

- $C_j(t)$ is its worst-case remaining execution time at speed $s_{\max}$,
- a_j is its arrival time,
- d_j is its absolute deadline.

Algorithm OPT makes the scheduling decision at time t by executing the first event in the queue at speed

$$s(t) = \max_{e_j} \left\{ \sum_{e_i : a_i \leq t, e_i \preceq e_j} \frac{C_i(t)}{d_j - t} \right\}. \tag{6.6}$$

In other words, the algorithm chooses the execution speed online in a greedy fashion and guarantees a low energy consumption by considering only the events that have arrived so far and are not yet completely processed.[2] Algorithm OPT has a competitive factor of γ^γ with respect to an optimal DVS schedule [BKP04]. This means that it consumes at most γ^γ as much dynamic energy as an optimal schedule which is computed offline, based on a priori knowledge of event arrival times.

---

[2] Note that if speed switching requires a timing overhead bounded by χ, we just have to modify $d_j - t$ in Equation (6.6) to $d_j - t - \chi$.

An important observation is that Algorithm OPT can guarantee the schedulability of any arbitrary event stream, provided that the system has no maximum speed constraint, that is, $s_{\max} = \infty$. However, if $s_{\max}$ is constrained, applying the algorithm can lead to a schedule that violates deadlines, because the algorithm may require execution speeds that exceed $s_{\max}$. In other words, the Algorithm OPT may be too optimistic in the sense that it is not provident enough. For this reason, we call the above online approach also *optimistic DVS scheduling*.

6.3 Design of Offline DVS Systems with MPA

In this section we consider the energy-efficient scheduling of multiple event-processing real-time tasks with static priorities and non-deterministic release times. We assume that the event arrival times are not known a priori but are constrained by arrival curves. For this setting, it is still an open issue how to statically or dynamically determine the execution speeds and the priorities of the individual tasks such that the energy consumption is minimized. We focus on offline algorithms, that is, on methods that fix the execution parameters at design time according to the predicted worst-case workload. Specifically, we explore how to statically determine the priorities and execution speeds of multiple tasks that are triggered by arbitrary event streams. We first consider systems with a bounded continuous speed range, and then adapt the developed concepts to the case of discrete operating speeds.

Determining the optimal individual execution speeds of tasks in a static priority setting and under arbitrary release patterns is not trivial. For instance, it is often not wise to process a high priority task as slowly as possible, as this might force low priority tasks to use very high speeds in order to meet their timing constraints. Also, there is a mutual dependency between priorities and execution speeds: a task can tolerate a lower execution speed if it has a higher priority. To the best of our knowledge, all present DVS methods for static priority systems consider task priorities as given. By not rearranging the priorities of tasks, they preclude an important possibility to further reduce the energy consumption.

The contributions of this section can be summarized as follows:

- We devise a simple algorithm for computing a static priority assignment to multiple real-time tasks with arbitrary release patterns. We show that the computed priority assignments are energy-optimal in the particular case that one global speed is used to execute all the tasks.

- We show that priority-monotonic speed assignments are energy-efficient, and present a heuristic that determines the individual task execution speeds in the particular case that the task priorities are predetermined.
- We propose an efficient heuristic for the general case in which both priorities and execution speeds need to be determined.
- We demonstrate the benefits of the presented methods in several experimental test cases.

6.3.1 Models and Problem Definition

In the following, we first describe the system and power models adopted in our approach. We then state the problem definition and provide a motivational example.

6.3.1.1 System Model

The studied system consists of a CPU that executes N independent real-time tasks, $\Gamma = \{\tau_1, \cdots, \tau_N\}$. The tasks are executed with static priorities and in a preemptive manner. The CPU supports DVS and hence can use different execution speeds (supply voltages) at different times. For the sake of simplicity, we assume that there is no timing or energy overhead for speed changes. We denote the set of available execution speeds with $\hat{\mathcal{S}}$. Recent DVS processors have only a discrete set of available speeds. However, it is still possible to emulate a continuous speed range by means of voltage hopping [LS00]. In our approach, we consider both the continuous and the discrete case, that is, we either assume $\hat{\mathcal{S}} = [\hat{s}_{min}, \hat{s}_{max}]$, or $\hat{\mathcal{S}} = \{\hat{s}_1, \hat{s}_2, \cdots, \hat{s}_K\}$ with $\hat{s}_{min} := \hat{s}_1,\ \hat{s}_{max} := \hat{s}_K,\ \hat{s}_1 < \hat{s}_2 < \cdots < \hat{s}_K$. In Section 6.3.2 we discuss both cases separately. Without loss of generality, we consider $\hat{s}_{max} = 1$ and normalize all related metrics.

Each task τ_i is specified by the following parameters:

- $\bar{\alpha}_i(\Delta)$: Upper arrival curve
- C_i : Execution time at the maximum speed $\hat{s}_{max}$
- D_i : Deadline (relative to the task release)

The arrival curve $\bar{\alpha}_i(\Delta)$ bounds the maximum number of task releases in any time interval of length Δ. We assume that all instances of a task τ_i have the same execution time C_i at the maximum speed $\hat{s}_{max}$. The curve $\alpha_i(\Delta) = C_i \cdot \bar{\alpha}_i(\Delta)$ represents an upper bound for the processing demand of τ_i in any time interval of size Δ. The processing demand is expressed in units of execution time at speed $\hat{s}_{max}$. If τ_i is executed at a different speed $s \in \hat{\mathcal{S}}$, the processing of an instance takes C_i/s time units. Hence, the curve $\alpha_i(\Delta)/s$ represents an upper bound for the demanded execution time in any interval of size Δ, under the assumption that the task is executed at speed s. The worst-case response time of a task τ_i is defined as the longest time between the release and the completion of the task. We say that τ_i is schedulable if $\mathrm{WCRT}\tau_i \leq D_i$.

6.3.1.2 Power Model

We consider the power model of [ZMM04], in which the power dissipation of the CPU at speed $\hat{s}$ is represented as follows:

$$P(\hat{s}) = P_{sta} + \hbar(P_{ind} + P_d) = P_{sta} + \hbar(P_{ind} + C_{ef}\hat{s}^{\gamma}) \tag{6.7}$$

In the above equation, P_{sta}, P_{ind}, and P_d are *static power*, *speed-independent* active power, and *speed-dependent* active power, respectively. If the system is in active mode, $\hbar$ is 1, whereas $\hbar$ is set to 0 when the system is in sleep mode. The constants C_{ef} and $2 \leq \gamma \leq 3$ are system-dependent, and represent the effective switching capacitance and the dynamic power exponent, respectively.

We exclude the possibility to turn the system off dynamically due to excessive time/energy overhead. This means that the static power P_{sta} is not manageable, and therefore we do not consider it in the reminder of the chapter. Instead, we focus on how to manage the active power dissipation of the CPU. We assume that in addition to the DVS carried out in the active mode, the processor can switch to a sleep mode in which there is no active power dissipation. In particular, we consider the simplest form of DPM in which the processor goes to the sleep mode whenever it is idle, and returns to active mode as soon as a task is released. We assume that there is no overhead in terms of time or energy for switching between the two modes. This is reasonable, if gated supply voltage is applied [ZMM04].

Under the mentioned assumptions, the dynamic energy consumption of the CPU is merely a convex function of the execution speed, and there is a *critical speed* $\hat{s}_{crit} \in \hat{\mathcal{S}}$ such that executing at $\hat{s}_{crit}$ is more energy-efficient than executing at any other (also lower) speed [JPG04, ZMM04]. By the definition of critical speed, only speeds in the set $\mathcal{S}$ should be used, where $\mathcal{S} = \{s \in \hat{\mathcal{S}} \mid s \geq \hat{s}_{crit}\}$. The set $\mathcal{S}$ can again be continuous, $\mathcal{S} = [s_{min}, s_{max}]$, or discrete, $\mathcal{S} = \{s_1, s_2, \cdots, s_M\}$ with $s_{min} := s_1,\ s_{max} := s_M$, and $s_1 < s_2 < \cdots < s_M$.

6.3.1.3 Problem Definition

We consider the problem of statically assigning priorities and execution speeds to real-time tasks with bounded non-deterministic release patterns, such that all tasks are guaranteed to meet their deadlines and the worst-case energy consumption of the system is minimized. Formally, we can define the problem as follows:

> Given is a set of real-time tasks $\Gamma = \{\tau_1, \cdots, \tau_N\}$ characterized as described above. Let $\Pi : \Gamma \rightarrow \{1, \cdots, N\}$ be a bijective function that assigns a unique priority to each task, where task τ_i has higher priority than task τ_j if $\Pi(\tau_i) < \Pi(\tau_j)$. Let $\Sigma : \Gamma \rightarrow \mathcal{S}$ be a function that assigns an execution speed to each task. Let $E(\Delta)$ denote the worst-case dynamic energy consumption of the system for any time interval of length Δ. The problem is to find static assignments Π and Σ such that the following two conditions hold:
>
> - $\mathrm{WCRT}\tau_i \leq D_i \quad \forall i \in \{1, \cdots, N\}$
> - $E(\Delta)$ is minimized

In the defined setting, each task is executed with a *constant* speed, that is, speed changes are allowed only at context switches. The constant execution speed of a task has to be chosen such that the worst-case workload of the task can always be handled. A more fine-grained DVS schedule in which different instances of the same task are executed with different speeds, or in which the speed is changed *during* the execution of a task (see e.g. [XMM04]) makes sense only in the presence of a priori knowledge of task release times, or in the context of online DVS.

Note that in general for different time intervals Δ_1, Δ_2 different static speed assignments Σ_1, Σ_2 can be optimal in terms of energy consumption, as we will show later. In other words, an overall optimal static speed assignment may not exist for a system. Therefore, we do not focus on optimizing the energy consumption for a specific time interval Δ, but rather devise heuristics that determine reasonable solutions for all sufficiently large intervals.

6.3.1.4 Motivational example

Let us now illustrate the above problem by means of a simple example. Consider a system with three tasks τ_1, τ_2, τ_3 that are released periodically with some non-deterministic but bounded release jitter. The tasks are specified by the parameter tuples $\tau_1 = \langle 10, 3, 1, 1\rangle$, $\tau_2 = \langle 5, 3, 1, 9\rangle$, $\tau_3 = \langle 8, 1, 1, 10\rangle$. Each tuple contains the parameters $\langle p, j, C, D\rangle$ (expressed in ms) with the following meaning: p = period, j = max. jitter, C = execution time at s_{max}, D = deadline. The active power consumption of the CPU is assumed to be $P(s) = \hbar(0.08 + 1.52s^3)$ Watt. We want to determine the static priority and speed assignments for the tasks that minimize the long-term worst-case energy consumption of the system.

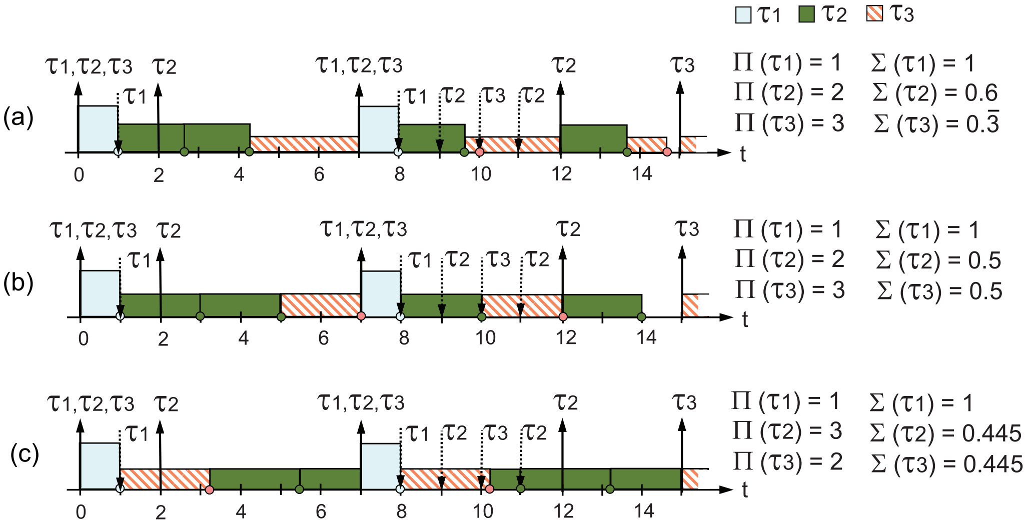


Fig. 71: Execution traces

Figure 71 shows the execution of the tasks under worst-case workload (critical instant) for three different priority and speed assignments. The upward and downward arrows in the figure indicate task releases and task deadlines, respectively. The rectangles represent the execution of task instances (also denoted as jobs). The height of a rectangle indicates the speed at which the corresponding job is executed. A small dot on the time axis indicates the completion of a job. If no dot is shown at the bottom right of a rectangle, it means that the corresponding job is preempted by a higher-priority job. The respective priority and speed assignments are shown next to the execution traces.

A feasible priority and speed assignment is shown in Figure 71(a). The corresponding execution trace shows that the first job of τ_3 completes just in time (at $t = 10$). We conclude that the speed assignment in (a) is 'tight', in the sense that lowering the execution speed for any of the three tasks leads to a deadline violation. Nevertheless, the speeds chosen in (a) are not the most energy-efficient ones. In order to illustrate this, we compute the worst-case dynamic energy consumption of the system for a time interval of 10s. By a simple simulation of the worst-case execution trace, we obtain an energy consumption of $E_a(10s) = 3.475J$. Consider now the speed assignment of Figure 71(b). It is again tight, as lowering any of the execution speeds would lead to a deadline violation for τ_3. For this case we compute a worst-case energy consumption of $E_b(10s) = 3.357J$, that is, assignment (b) is more energy-efficient than (a). Finally, we decide to invert the priorities of τ_2 and τ_3. Figure 71(c) shows that this permits to use even lower speeds compared to (b), while the real-time constraints are still guaranteed. The corresponding worst-case energy consumption amounts to $E_c(10s) = 3.163J$. In the following, we discuss heuristics in order to find such energy-efficient priority and speed assignments.

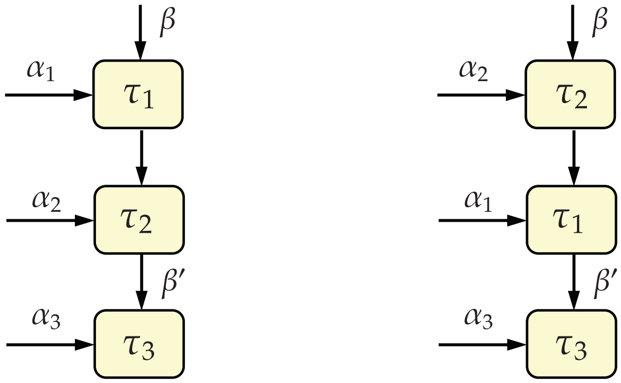
6.3.2 Proposed Algorithms

In this section, we present the algorithms that we propose to solve the static priority and speed assignment problem with. For the sake of clarity, we first discuss the assignments of priorities and speeds in isolation (Sections 6.3.2.1 and 6.3.2.2). Thereafter, we show how the algorithms can be integrated to achieve more energy-efficient solutions (Section 6.3.2.3). We initially consider systems with continuous speeds, and then explain how to adapt the algorithms for the discrete case (Section 6.3.2.4).

6.3.2.1 Priority Assignment

Let us first consider the simple case where we want to determine the priority assignment Π for a task set Γ with *fixed* speed assignment Σ. Since the execution speeds for the tasks are fixed, so also is the worst-case energy consumption of the system. This is because the amount of energy consumed depends only on the speed at which the single tasks are executed, but not on the order in which the tasks are processed. Nevertheless, the priorities of the tasks have an influence on the schedulability of the system. We want to determine a priority assignment Π under which Γ is schedulable, if such an assignment exists. Note that for N tasks, there are $N!$ possible priority assignments, and a brute-force algorithm would have to check all of them in the worst case. For simplified instances of this problem, one can find efficient solutions in the literature. For instance, the deadline-monotonic (DM) priority assignment has been shown to be optimal for periodically activated tasks with relative deadlines smaller than the respective periods [LW82]. In the following, we devise a simple and efficient algorithm that solves the priority assignment problem in the general case, that is, under arbitrary task activation patterns.

The basic idea behind our method is that the processing resources which are available to a task do not depend on the order in which higher-priority tasks are processed. This is illustrated in the example of Figure 72, where the input service curve β' of the lowest-priority task τ_3 is the same in both shown cases. Therefore, for verifying whether a given task is schedulable at the lowest priority level, it is sufficient to perform a single run of the scheduling analysis, with an arbitrary arrangement of the



(a) (b)

Fig. 72: Two chains of Greedy Processing Components with different priority assignments but same service for τ_3

Algorithm 2 `Determine priorities`

```
function FIND_SCHEDULABLE_ORDER($\Gamma, \Sigma, \beta$): $\Pi$
    $\Pi \leftarrow$ random priority assignment on $\Gamma$
    for $i \leftarrow N, 1$ do
        last_task_schedulable $\leftarrow$ false
        for $j \leftarrow 1, i$ do
            $\Pi' \leftarrow \Pi$
            $\Pi'(\Pi^{-1}(i)) \leftarrow j$
            $\Pi'(\Pi^{-1}(j)) \leftarrow i$
            $\beta' = \beta$
            for $k \leftarrow 1, i - 1$ do
                $\tau_x \leftarrow \Pi'^{-1}(k)$
                $\beta' = \mathrm{RT}(\beta', \alpha_x/\Sigma(\tau_x))$
            end for
            $\tau_x \leftarrow \Pi'^{-1}(i)$
            if $\mathrm{Del}(\alpha_x/\Sigma(\tau_x), \beta') \leq D_x$ then
                last_task_schedulable $\leftarrow$ true
                $\Pi \leftarrow \Pi'$
                break
            end if
        end for
        if not last_task_schedulable then
            Warning($\Gamma$ NOT SCHEDULABLE)
            return $\perp$
        end if
    end for
    return $\Pi$
end function
```

higher-priority tasks. This concept is employed iteratively in Algorithm 2 to derive a schedulable priority assignment Π for a task set Γ that is executed on a CPU with service β. The algorithm goes through all priority levels, starting from the lowest, and assigns a task to each level. The task is chosen among the unassigned ones by making sure that it is schedulable at the given priority level. In the algorithm, we make use of the inverse priority assignment function $\Pi^{-1}(\gamma)$, which for a given priority level γ returns the task assigned to it. The time complexity of Algorithm 2 is $O(N^3)$. The following theorem states the optimality of the described priority assignment strategy.

Thm. 13: *Let Γ be a task set specified as described above and Σ a speed assignment for Γ. Let β be the service guaranteed by a CPU with preemptive static priority scheduling. This being the case, Algorithm 2 finds a priority assignment Π such that the system $\langle \Gamma, \Pi, \Sigma, \beta \rangle$ is schedulable, provided that such a priority assignment exists.*

Proof. The theorem can be shown by contradiction. Assume that $\exists\, \Pi$ such that $\langle \Gamma, \Pi, \Sigma, \beta \rangle$ is schedulable, but Algorithm 2 does not return Π. This means that the algorithm either returns another schedulable priority assignment Π', which is fine, or it returns $\perp$. In the latter case, the algorithm must have reached a priority level γ such that none of the γ unassigned tasks can tolerate the execution at the lowest unassigned priority level. Note that revising priority assignments done previously is not helpful, as this would comport an even lower priority for at least one of these γ tasks. Hence, Algorithm 2 returns $\perp$ only in the case where there is no schedulable priority assignment for the system.

□

The above algorithm can be extended to solve another simplified instance of the problem described in Section 6.3.1.3: Find the most energy-efficient pair (Π, Σ) for a task set Γ, under the restriction that all tasks execute at *equal* speed. In other words, determine the priority assignment that allows the minimum *global* execution speed. The proposed solution for this problem is shown in Algorithm 3. We will reuse it within the combined optimization heuristic, discussed in Section 6.3.2.3. In Algorithm 3, we use Σ_s to denote the assignment of the same speed s to all tasks in Γ. The algorithm implements a simple binary search strategy, in which the global speed is increased if no schedulable priority assignment is found or decreased otherwise. The search stops once the minimum global speed is approximated with a specified precision ϵ. Algorithm 3 has time complexity $O(N^3 \log \frac{1}{\epsilon})$.

Algorithm 3 `Determine min global speed`

```
function MIN_GLOBAL_SPEED($\Gamma, \beta, \mathcal{S}, \epsilon$): $s$, $\Pi$
    $\Pi \leftarrow$ Find_schedulable_order ($\Gamma, \Sigma_{s_{max}}, \beta$)
    if $\Pi = \perp$ then
        Error($\Gamma$ NOT SCHEDULABLE)
    end if
    $s_{up} \leftarrow s_{max}$
    $s_{lo} \leftarrow s_{min}$
    repeat
        $s \leftarrow (s_{lo} + s_{up})/2$;
        $\Pi' \leftarrow$ Find_schedulable_order ($\Gamma, \Sigma_s, \beta$)
        if $\Pi' = \perp$ then
            $s_{lo} \leftarrow s$
        else
            $s_{up} \leftarrow s$
            $\Pi \leftarrow \Pi'$
        end if
    until $s_{up} - s_{lo} \leq \epsilon$
    return $s_{up}$, $\Pi$
end function
```

Cor. 3: *Let Γ be a task set specified as described above. Let β be the service guaranteed by a processor with preemptive static priority scheduling and let $\mathcal{S} = [s_{min}, s_{max}]$ denote the continuous set of available execution speeds. Assume that a priority assignment Π and a* minimal *speed $s \in \mathcal{S}$ exist such that $\langle \Gamma, \Pi, \Sigma_s, \beta \rangle$ is schedulable. Let $\epsilon > 0$ be an arbitrary small constant. This being the case, Algorithm 3 finds a priority assignment Π' and a speed $s' \in \mathcal{S}$ with $s' - s \leq \epsilon$ such that the system $\langle \Gamma, \Pi', \Sigma_{s'}, \beta \rangle$ is schedulable.*

Proof. Follows directly from Theorem 13 and the monotonicity of the schedulability with respect to the execution speed: A task set Γ which is schedulable at a global speed s is also schedulable at a global speed $\tilde{s} > s$.

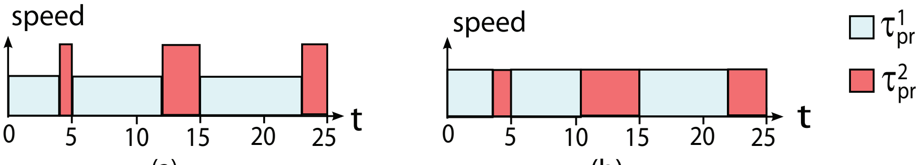
□

Since the energy consumption function is monotone on $\mathcal{S}$, we conclude that Algorithm 3 finds the energy-optimal *global* speed assignment that satisfies the timing constraints.

6.3.2.2 Speed Assignment

Let us now consider the opposite case where we want to determine an energy-efficient speed assignment Σ for a task set Γ with *fixed* priority assignment Π. For better readability, in the following we make use of the abbreviated notation $\tau_{pr}^{i} := \Pi^{-1}(i)$ to refer to the task assigned to a priority level i. We denote the task parameters accordingly: $\alpha^i_{pr}, D^i_{pr}, \cdots$. Analogously, we denote the execution speed of the task at priority level i with $s^i_{pr} := \Sigma(\Pi^{-1}(i))$, and the service available to that task with β^i_{pr}.

For minimizing the energy consumption for the execution of multiple tasks, it is typically not adequate to assign the same execution speed to all the tasks, as done in Section 6.3.2.1. For instance, the minimum admissible global speed for a priority chain of tasks could be determined by a high priority task with a stringent deadline. However, this speed could be unnecessarily high for tasks with lower priorities and less stringent timing requirements. To save energy, we should further slow down lower priority tasks as much as possible, and avoid premature task completions. In other words, we should always use *tight* speed assignments. We call a speed assignment tight if slowing down any of the speeds in the assignment compromises the timing guarantees. Unfortunately, there are often a multitude of different tight speed assignments for a chain of tasks. Hence, it is not easy to find the most energy-efficient one. This is because the assignment of a speed to a particular priority level affects the speeds required at lower priority levels. For instance, it may not be energy-efficient to process a high priority task slowly, if in turn this forces a low priority task to use a high speed to meet its deadline. The convexity of the power function suggests using the same execution speed for both tasks in such a situation, or, if two different speeds are necessary, as close speeds as possible. This is illustrated in the example of Figure 73, which shows two different execution traces for a system with two tasks. Trace (a) results from an assignment with $s^1_{pr} < s^2_{pr}$ where we assume that the assignment is tight with respect to the deadline of τ^2_{pr}. Trace (b) results from another tight assignment with $s^1_{pr} = s^2_{pr}$. This second assignment is more energy-efficient due to the convexity of the power function.



(a) (b)

Fig. 73: Schedules in a busy interval. (a) Original schedule. (b) Lower energy consumption.

Based on the above observation, we propose using speed assignments Σ that satisfy the following constraints:

$$s^1_{pr} \geq s^2_{pr} \geq \cdots \geq s^N_{pr} \tag{6.8}$$

$$s^{i-1}_{pr} > s^i_{pr} \text{ only if WCRT } \tau^{i-1}_{pr} = D^{i-1}_{pr} \tag{6.9}$$

The first constraint imposes a *priority-monotonic* speed assignment, such that a lower priority task never executes at a higher speed than a higher priority task. The second constraint enforces the use of as close speeds as possible. It states that a high priority task is executed at a higher speed than a low priority task only in case it cannot be further slowed down. Note that priority-monotonic speed assignments have also been explored in [SR03], however for strict periodic tasks only.

Let us now introduce a heuristic to compute an energy-efficient speed assignment Σ that fulfills the above constraints. The basic principle of the heuristic is first, to determine a minimum *global* speed assignment, and then to find the 'bottleneck' of the assignment. The bottleneck is the highest priority task τ_{pr}^{i} that cannot tolerate a lower speed assignment for any task τ_{pr}^{j} with $j \leq i$, as this would compromise its schedulability. At this point the speed assignments for $\tau_{pr}^{1}, \cdots, \tau_{pr}^{i}$ are made definitive and the procedure is repeated for the lower priority tasks. The procedure stops once τ_{pr}^{N} becomes the bottleneck of a speed assignment.

Algorithm 4 implements the described procedure. The function `MinSpeed` employs a binary search to find the minimum global speed for the remaining part of the task chain, given the service β_{in} left over by higher priority tasks. The scheduling analysis for the respective tasks is carried out by the function `Sched`, which implements Equation (2.41). The function `FindBottleneck` determines the bottleneck of a given sub-chain of tasks, with guaranteed service β_{in} and under constant speed s. The overall time complexity of Algorithm 4 is $O(N^2 \log \frac{1}{\epsilon})$.

Remarks on Non-Optimality

We mentioned already in the problem description that for different time intervals, different static speed assignments can be optimal in terms of energy consumption. Therefore, we cannot expect that the above algorithm always determines an overall optimal speed assignment. The following example highlights the potential non-optimality of the proposed speed assignment strategy.

Ex. 2: *Consider the execution of two periodic tasks that are specified by the parameter tuples* $\tau_{pr}^{1} = \langle 8, 0, 1, 2 \rangle$ *and* $\tau_{pr}^{2} = \langle 4, 0, 1, 4 \rangle$ *where we use again the format* $\langle p, j, C, D \rangle$. *Assume that* τ_{pr}^{1} *is executed at a general speed* $s_{pr}^{1} = x \geq 0.5$, *as shown in the worst-case arrival trace represented in Figure 74. Consequently,* τ_{pr}^{1} *needs to be executed at speed* $s_{pr}^{2} = \frac{1}{4-1/x}$ *in order to meet its deadline. Assume that the speed-dependent power dissipation of the system is given by* $P(s) = s^3$. *Then, the energy consumption of the system for the interval* Δ_2 *shown in Figure 74 amounts to* $E_x(\Delta_2) = \frac{1}{x} \cdot x^3 + 2 \cdot (4 - \frac{1}{x}) \cdot (\frac{1}{4-1/x})^3$, *which is minimal for* $x = 0.565$. *With Algorithm 4 we would use* $x = 0.5$, *that is, execute both*

Algorithm 4 `Determine speeds` (part 1)

```
function DETERMINE_SPEEDS($\Gamma, \Pi, \beta, \mathcal{S}, \epsilon$): $\Sigma$
    if SCHED($\Gamma$, $\Pi$, 1, $n$, $s_{max}$, $\beta$) = true then
        $s^0_{pr} \leftarrow s_{max}$
        $\beta^1_{pr} \leftarrow \beta$
        $i \leftarrow 0$
        repeat
            $i \leftarrow i + 1$
            $s \leftarrow$ MINSPEED ($\Gamma$, $\Pi$, $i$, $N$, $s_{min}$, $s^{i-1}_{pr}$, $\beta^i_{pr}$, $\epsilon$)
            for $k \leftarrow i, N$ do
                $s^k_{pr} \leftarrow s$
                $\beta^{k+1}_{pr}$ = RT($\beta^k_{pr}$, $\alpha^k_{pr}/s$)
            end for
            if $s - s_{min} < \epsilon$ then
                break
            else
                $i \leftarrow$ FINDBOTTLENECK ($\Gamma$, $\Pi$, $i$, $s$, $\beta^i_{pr}$, $\epsilon$)
            end if
        until $i = N$
    else
        ERROR(UNSCHEDULABLE)
    end if
    return $\Sigma = \{\tau^1_{pr} \rightarrow s^1_{pr}, \cdots, \tau^N_{pr} \rightarrow s^N_{pr}\}$
end function

function MINSPEED($\Gamma$, $\Pi$, From_pr, To_pr, $s_{lo}$, $s_{up}$, $\beta_{\text{in}}$, $\epsilon$): $s$
    repeat
        $s \leftarrow (s_{lo} + s_{up})/2$;
        if SCHED($\Gamma$, $\Pi$, From_pr, To_pr, $s$, $\beta_{\text{in}}$) = true then
            $s_{up} \leftarrow s$;
        else
            $s_{lo} \leftarrow s$;
        end if
    until $s_{up} - s_{lo} \leq \epsilon$
    return $s_{up}$
end function
...
```

Algorithm 4 `Determine speeds` (part 2)

```
    ...
function SCHED($\Gamma, \Pi$, From_pr, To_pr, $s, \beta_{\text{in}}$): schedulable
    $\beta' = \beta_{\text{in}}$
    for $k \leftarrow$ From_pr, To_pr do
        if $\text{Del}(\alpha^k_{pr}/s, \beta') \le D^k_{pr}$ then
            $\beta' = \text{RT}(\beta', \alpha^k_{pr}/s)$
        else
            return false
        end if
    end for
    return true
end function

function FINDBOTTLENECK($\Gamma, \Pi$, From_pr, $s, \beta_{\text{in}}, \epsilon$): $pr$
    $s_{red} \leftarrow s - \epsilon$
    for $k \leftarrow$ From_pr, $N$ do
        if SCHED ($\Gamma$, $\Pi$, From_pr, $k$, $s_{\text{red}}$, $\beta_{\text{in}}$) = false then
            return $k$
        end if
    end for
end function
```

tasks at the same speed $s^1_{pr} = s^2_{pr} = 0.5$*. This second speed assignment is clearly not optimal for* Δ_2 *but it minimizes the energy consumption in other intervals, e.g.,* Δ_1.

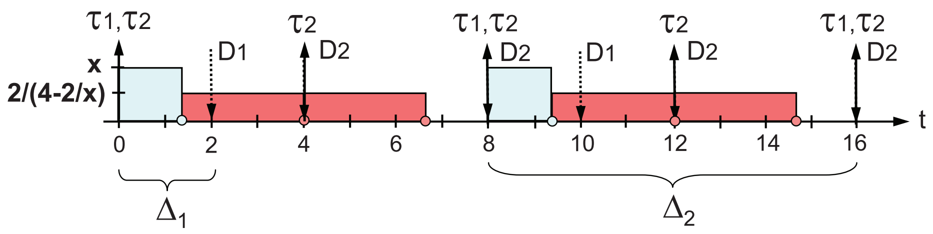


Fig. 74: Execution trace for example system

The above example demonstrates that the constraint (6.9), which enforces the use of as close speeds as possible, does not always lead to the most energy-efficient solution. Nevertheless, the heuristic gives reasonable solutions for sufficiently large time intervals, as we will demonstrate in Section 6.3.3.

6.3.2.3 Combined Priority and Speed Assignment

Let us now consider the general case where we want to determine both the priority assignment Π and the speed assignment Σ for a task set Γ. For this case, we propose a heuristic that combines Algorithms 4 and 3. In particular, we integrate the priority reordering in the bottleneck speed assignment algorithm. At each loop iteration of the bottleneck algorithm, instead of just revising the global speed for a task sub-chain, we reorder the tasks in the sub-chain such that the minimal execution speed is achieved. In other words, we alternate task reordering and bottleneck finding in task sub-chains. This results in an algorithm with time complexity $O(N^4 \log \frac{1}{\epsilon})$.

6.3.2.4 Adaptations for Discrete Speeds

Practical CPUs typically provide a finite number of operating speeds/frequencies. In this section, we illustrate how the above algorithms can be adapted to the discrete case in which $\mathcal{S} = \{s_1, s_2, \cdots, s_M\}$.

Algorithms 2 and 3 can be easily reproduced for the discrete case. The only adaptation in Algorithm 3 concerns the binary search over $\mathcal{S}$, which is now carried out over the discrete set of available speeds. The time complexity of this discrete version of Algorithm 3 is $O(N^3 \log M)$, where M is the number of available execution speeds. The reformulation of Algorithm 4 for the discrete case is less trivial, as there is no clear notion of bottleneck task anymore. In particular, if after a global speed assignment to a sub-chain of tasks a particular task τ_{pr}^i results unschedulable, it could be overly conservative to execute all tasks τ_{pr}^j of the sub-chain with $j \leq i$ at the next highest speed. Rather, it could suffice to increase the speed of just a few of those tasks to guarantee the schedulability of τ_{pr}^i. A simple workaround to this problem is to treat each priority level as bottleneck, meaning that we repeat the speed assignment to the remaining tasks at each priority level. The corresponding pseudo-code is shown in Algorithm 5. For better readability, in Algorithm 5 we denote a discrete speed s_i with $s(i)$. To find minimum global speeds for sub-chains of tasks, the algorithm employs the function `MinSpeedLevel`, which implements a binary search on the ordered set of discrete speeds. The time complexity of Algorithm 5 is $O(N^2 \log M)$ in contrast to the complexity $O(M^N)$ of a brute-force solution.

Finally, as done in the continuous case, we combine the algorithms for priority and speed assignment. In particular, each time we have to assign a global speed to a sub-chain of tasks, we reorder the tasks such that the minimum speed is obtained. Note that in the resulting heuristic, we still reorder the tasks assuming *continuous* speeds, and then assign the next highest available discrete speed. This choice is done because

Algorithm 5 `Determine discrete speeds`

```
function DETERMINE_DISCRETE_SPEEDS($\Gamma, \Pi, \beta, \mathcal{S}$): $\Sigma$
    if SCHED($\Gamma$, $\Pi$, 1, $n$, $s_M$, $\beta$) = true then
        $sl_{pr}^{0} \leftarrow M$
        $\beta_{pr}^{1} \leftarrow \beta$
        for $i \leftarrow 1, N$ do
            $sl \leftarrow$ MINSPEEDLEVEL ($\Gamma$, $\Pi$, $\mathcal{S}$, $i$, $N$, 1, $sl_{pr}^{i-1}$, $\beta_{pr}^{i}$)
            for $k \leftarrow i, N$ do
                $sl_{pr}^{k} \leftarrow sl$
                $\beta_{pr}^{k+1} = \mathrm{RT}(\beta_{pr}^{k}, \alpha_{pr}^{k}/s(sl))$
            end for
            if $sl = 1$ then
                break
            end if
        end for
    else
        ERROR(UNSCHEDULABLE)
    end if
    return $\Sigma = \{\tau_{pr}^{1} \rightarrow s(sl_{pr}^{1}), \cdots, \tau_{pr}^{\mathrm{N}} \rightarrow s(sl_{pr}^{\mathrm{N}})\}$
end function

function MINSPEEDLEVEL($\Gamma, \Pi, \mathcal{S}$, From_pr, To_pr, $sl_{lo}, sl_{up}, \beta_{\mathrm{in}}$): $sl$
    if SCHED($\Gamma$, $\Pi$, From_pr, To_pr, $s_1$, $\beta_{\mathrm{in}}$) = true then
        $sl \leftarrow 1$
        return $sl$
    end if
    repeat
        $sl \leftarrow \lceil (\mathrm{sl}_{lo} + \mathrm{sl}_{up})/2 \rceil$;
        if SCHED($\Gamma$, $\Pi$, From_pr, To_pr, $s(sl)$, $\beta_{\mathrm{in}}$) = true then
            $sl_{up} \leftarrow sl$
        else
            $sl_{lo} \leftarrow sl$
        end if
    until $sl_{up} - sl_{lo} = 1$
    return $sl_{up}$
end function
```

under discrete speeds the reordering procedure is less selective (assuming $\epsilon << \left|s_i - s_j\right| \; \forall s_i, s_j \in \mathcal{S}_{discr}$). In other words, under discrete speeds the reordering procedure is more likely to determine a priority order that restrains further speed reductions at lower priority levels. The described heuristic has time complexity $O(N^4 \log \frac{1}{\epsilon})$, compared to $O(N!M^N)$ of a brute-force approach.

6.3.3 Experimental Evaluation

In this section, we experimentally evaluate the quality of the proposed algorithms. We compare the worst-case energy consumption of different systems when the priorities and the execution speeds of tasks are determined according to Algorithms 3, 4, 5, and the combined heuristics. We consider both continuous and discrete sets of execution speeds, and compare the corresponding results.

6.3.3.1 Experimental setup

We have implemented the above algorithms in Matlab. The algorithms call functions of the RTC Toolbox to handle arrival and service curves and perform the scheduling analysis. We construct nine test cases to evaluate the algorithms. Each test case consists of a processor that executes ten tasks with static priorities and preemptive scheduling. Each task is activated by an event stream which is represented by means of an arrival curve. To facilitate the replication of the experiments by means of other analysis tools, we use PJD models for the timing characterization of the event streams. For each test case, we randomly generate integer parameters for ten tasks, as shown in Table 10 for test case 6. The periods p_i are chosen uniformly in the range $[5\text{ms}, 30\text{ms}]$, the jitters j_i in $[0, 2p_i]$, the minimum event inter-arrival times d_i in $[0, \lceil p_i/4 \rceil]$, and the execution times C_i in $[1\text{ms}, \lceil p_i/15 \rceil]$. We also associate a relative deadline D_i to each task.

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6	τ_7	τ_8	τ_9	τ_{10}
p	14	28	15	30	28	7	25	25	30	22
j	20	35	3	19	35	5	30	20	0	15
d	2	2	0	0	4	0	5	0	0	0
C	1	1	1	1	1	1	2	1	1	2
D	21	1	53	406	276	41	6	342	4	181

Tab. 10: Parameters for test case No. 6 [ms]

For each test case, we automatically construct an MPA model and discard parameter sets that are not schedulable at the maximum speed.

The arrival curves are obtained with $\alpha_i(\Delta) = C_i \cdot \bar{\alpha}_i(\Delta)$, where we determine $\bar{\alpha}_i(\Delta)$ according to Equation (2.13). We let β represent a fully available CPU, and set the precision requirement for the case of continuous speeds to $\epsilon = 10^{-4}$. The complete parameter sets for the nine test cases, the full list of computed results, as well as the source code of the algorithms are available online.[3]

For determining the worst-case energy consumption of the various test systems, we have implemented a simple discrete-event simulator. The simulator reproduces worst-case task activation patterns, simulates the execution of tasks, and keeps track of the consumed energy. We assume a power dissipation of $\hbar(0.08 + 1.52s^3)$ *Watt*. This function approximates the active power dissipation of an Intel XScale CPU with maximum frequency of 1GHz normalized to $s_{\max} = 1$.

For each test case we consider two scenarios: A) continuous speed assignment, B) discrete speed assignment. For the two scenarios, we assume $\hat{\mathcal{S}}_A = [0, 1]$ and $\hat{\mathcal{S}}_B = \{0.15, 0.4, 0.6, 0.8, 1\}$, respectively. After computing the critical speed, we obtain $\mathcal{S}_A = [0.297, 1]$ and $\mathcal{S}_B = \{0.4, 0.6, 0.8, 1\}$. For each test case we consider three different priority and speed assignment policies, and compare the corresponding worst-case energy consumption for a time interval $\Delta = 10^4$ ms. For scenario A (continuous speeds) the policies are as follows:

(a) Compute Π and Σ_s with Algorithm 3

(b) Compute Π' with Algorithm 3, then compute Σ' with Algorithm 4

(c) Compute Π'' and Σ'' with the combined heuristic

For scenario B (discrete speeds), the priority and speed assignment policies are defined accordingly.

6.3.3.2 Results

Table 11 reports the detailed priority and speed assignments derived by the various algorithms for test case 6 and scenario A (continuous speeds). The upper line in the rows indicates which task has been assigned to the corresponding priority level. The lower line shows the execution speed assigned to the task. The table illustrates that a global speed assignment may be pessimistic for a set of real-time tasks with static priorities. This is because the execution speeds of some tasks can be reduced without harming any timing constraints. In addition, the table shows that the combined optimization of priorities and speeds can lead to considerably better results than consecutive priority and speed assignments.

---

[3] `http://www.tik.ee.ethz.ch/%7Epsimon/FPDVS.zip`

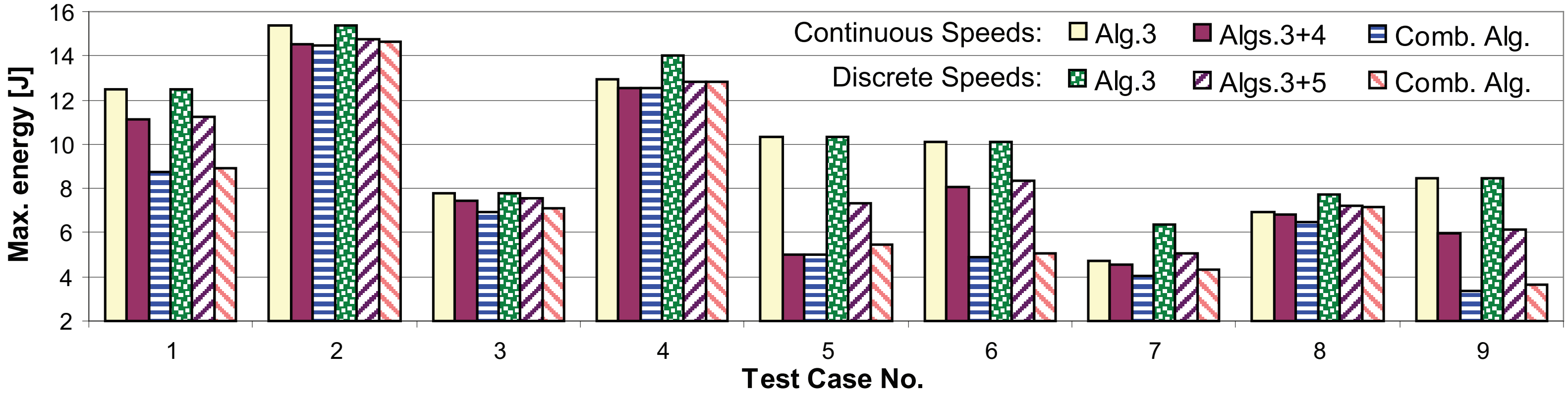


Fig. 75: Worst-case energy consumption for an interval of $\Delta = 10^4$ ms.

Policy \Priority	1	2	3	4	5	6	7	8	9	10
(a) Algorithm 3	τ_2 1.000	τ_9 1.000	τ_7 1.000	τ_{10} 1.000	τ_8 1.000	τ_6 1.000	τ_1 1.000	τ_5 1.000	τ_4 1.000	τ_3 1.000
(b) Algorithms 3 + 4	τ_2 1.000	τ_9 .9445	τ_7 .9445	τ_{10} .9445	τ_8 .9445	τ_6 .9445	τ_1 .9445	τ_5 .5776	τ_4 .5776	τ_3 .5776
(c) Combined	τ_2 1.000	τ_9 .7500	τ_7 .7500	τ_8 .5914	τ_1 .5914	τ_6 .5914	τ_3 .5914	τ_5 .5914	τ_4 .5914	τ_{10} .5914

Tab. 11: Priority and speed assignments determined by the different approaches for test case 6 and scenario A

Figure 75 sums up the the worst-case energy consumption for all nine test cases and all assignment policies. The chart shows that the conclusions of test case 6 also apply to the remaining test cases. In particular, the heuristic for combined priority and speed assignment clearly outperforms the other algorithms. It achieves energy savings up to 60% compared to the optimal priority ordering for minimum global speed (Algorithm 3), and up to 44 % compared to the consecutive priority and speed assignment (Algorithms 3 + 4). The results demonstrate that a large part of the optimization potential is sacrificed if priorities and speeds are not optimized in a combined manner. The chart of Figure 75 also compares the worst-case energy consumption under the discrete speed set $\mathcal{S}_B$, and the continuous speed set $\mathcal{S}_A$. The comparison shows that in the discrete case, more energy is consumed due to the coarser granularity of the speed assignments.

Scenario	Continuous (A)			Discrete (B)		
Assignment Policy	(a)	(b)	(c)	(a)	(b)	(c)
Minimum run-time	3.4	4.5	7.8	0.6	1.2	15.6
Maximum run-time	43.5	49.6	56.4	6.3	10.0	357.5

Tab. 12: Minimum and maximum run-times measured for the different assignment policies [seconds].

In Table 12 we report the minimum and maximum run-times measured for the different algorithms. For policy (b) the cumulative run-time of both employed algorithms is indicated. The table shows that for the considered test cases with ten tasks the algorithms compute the priority and speed assignments in less than one minute, and that in general the algorithms for the discrete setting are considerably faster than their continuous counterparts. This is simply because, in the more coarse-grained discrete setting, less solutions have to be explored. Nevertheless, the combined heuristic is considerably slower in the discrete case compared to the continuous one, which might seem incongruous. The reason for this effect is that in both the continuous and the discrete scenarios, continuous speeds are used for the task reordering procedure (cf. Section 6.3.2.4), but Algorithm 5 typically explores more cases than Algorithm 4, because it repeats the task reordering at *each* priority level.

6.4 Design of Online DVS Systems with TA

In the previous section, we concentrated on minimizing the energy consumption by finding the minimum (or most effective) *constant* speeds to process individual event streams. This is also the approach followed in [MCT05] for a single input event stream. However, as the arrival curve representing a stream does not reveal how events *actually* arrive, this constant speed must be dimensioned for the worst-case. Ordinary event arrival patterns will, in general, not lead to the worst-case workload for the processor. Hence, when applying offline DVS scheduling for the execution of non-deterministic streams, most of the time the system will run at an unnecessarily high speed. For instance, if bursts of events happen only rarely in a stream, the above methods would, nevertheless, always run at a high speed in order to guarantee schedulability.

Instead of choosing a constant speed assignment, one can dynamically adapt the execution speed to the current workload by means of online DVS algorithms. These methods decide the execution speed, based on the events that actually arrive on the input. Various online DVS algorithms can be found in the literature [YDS95, BP05]. They guarantee the schedulability of arbitrary input event streams, provided that there is no upper limit for the CPU speed. Obviously, this is not a reasonable assumption for real CPUs. In other words, it may happen that the execution speed required by an online DVS algorithm exceeds the maximum speed of a CPU, which translates to deadline violations. This imposes a careful verification of the maximum required speed before an online DVS algorithm can be applied [CST09]. However, an online DVS algorithm might exceed the maximum available speed only for a short time interval, depending on the actual event arrivals. Falling back on a constant speed assignment in such a case, as suggested by Chen et al. [CST09], is overly pessimistic.

In this section we propose an adaptive DVS scheme, in which an online DVS algorithm is applied when the system is *light-loaded* and a pessimistic speed is assigned when the system is *heavy-loaded*. On one hand, the proposed adaptive DVS algorithm reduces the energy consumption by being as optimistic as possible. On the other hand, it guarantees the schedulability of the event stream by switching to a pessimistic mode once the system is heavily loaded. The key issue for the adaptive DVS algorithm is to decide when to be pessimistic and when to be optimistic. In the following, we discuss how to design and analyze such an adaptive DVS scheme by means of Timed Automata.

6.4.1 System Model and Problem Definition

The studied system consists of a CPU with DVS that supports a continuous range of execution speeds $\hat{\mathcal{S}} = [\hat{s}_{min}, \hat{s}_{max}]$. We assume that the power dissipation of the CPU can be abstracted by the general power model of Equation (6.7). As already done in Section 6.3, we discard execution speeds below the critical speed $\hat{s}_{crit}$, that is, we restrict the selectable speeds to the range $\mathcal{S} = [s_{min}, s_{max}]$.

The CPU executes a single real-time task τ which is triggered by the events of a non-deterministic input event stream. The event stream is characterized by an upper arrival curve $\bar{\alpha}(\Delta)$ which specifies the maximum number of possible task activations in a time interval of length Δ. We assume that τ has an execution time C at maximum speed s_{max} and a relative deadline D. We again use a curve $\alpha(\Delta) = C \cdot \bar{\alpha}(\Delta)$ to express the maximum processing demand imposed on the system, where the processing demand is expressed in units of execution time at speed s_{max}. If the CPU runs at a different speed $s \in \mathcal{S}$, then the execution time requested in a time interval of length Δ amounts to $\alpha(\Delta)/s$.

For the above system, we consider the problem of finding a dynamic DVS schedule such that all deadlines are met, and, at the same time, the energy consumption of the CPU is minimized.

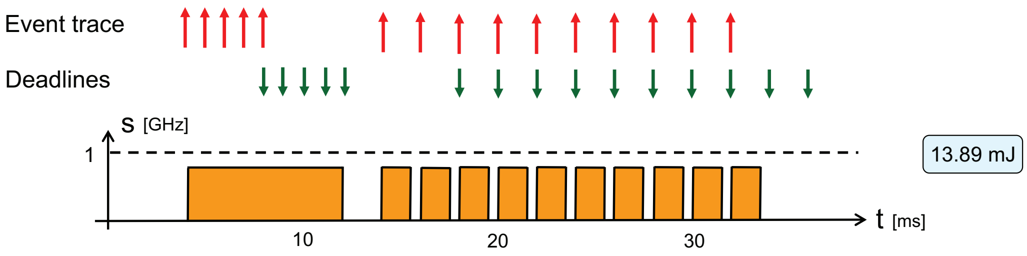
6.4.2 Adaptive Scheduling Scheme

In this section, we present a concrete example that motivates the application of an adaptive DVS scheme, followed by the definition of such a scheduling scheme.

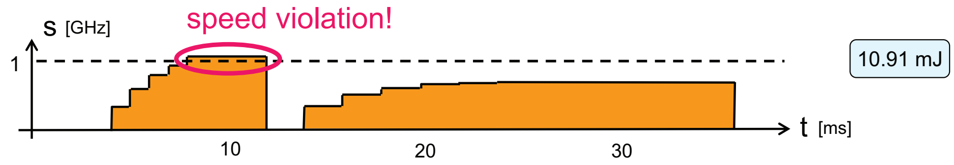
6.4.2.1 Motivational Example

Consider an input event stream specified by a PJD event model with parameters $p = 2$ ms, $j = 4$ ms, and $d = 1$ ms. The corresponding upper arrival curve $\bar{\alpha}(\Delta)$ can be obtained by means of Equation (2.13). Assume that the CPU can select execution speeds in the range $s_{min} = 0$, $s_{max} = 1\,\text{GHz}$ and that its power dissipation is described by the simple function $P(s) = \hbar(\frac{s}{1GHz})^3$ Watt. The processing time of an input event is $\frac{4}{3}$ ms at speed s_{max} and the relative deadline D is 4 ms.

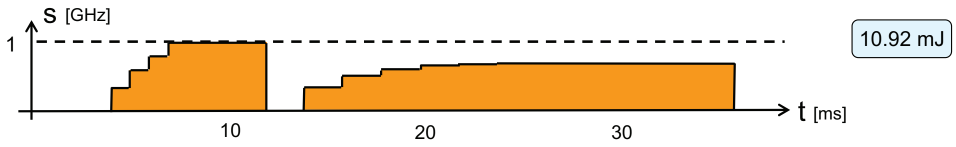
Let us now compare the behaviour of three different DVS schemes in the presence of a concrete input event trace that conforms to the above



(a) Static DVS



(b) Dynamic DVS



(c) Adaptive DVS

Fig. 76: Execution traces of different DVS schedulers

model. We consider the input trace r consisting of 15 events with arrival times $(4, 5, 6, 7, 8, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32)$ ms.

1. The first DVS scheme is an offline scheduler that statically determines a constant execution speed for $\bar{\alpha}$ according to Equation (6.5). The resulting constant execution speed is $s_{SD} = \frac{5}{6}$ GHz. The corresponding execution of r is shown in Figure 76(a) and leads to an energy consumption of 13.89 mJ.

2. The second DVS scheme is an online scheduler that implements Algorithm OPT, as described in Section 6.2.2. This scheduler adapts the execution speed dynamically as represented in Figure 76(b). The corresponding energy consumption amounts to 10.91 mJ, that is, the dynamic method is more energy-efficient than the static one. However, Figure 76(b) reveals that the maximum speed required by Algorithm OPT for the execution of r exceeds s_{max}. In other words, the online DVS scheduler cannot be used to ensure the timely execution of the input stream. Specifically, in the time interval [8 ms, 12 ms] Algorithm OPT requires an execution speed of 1.017 GHz.

3. Looking at the execution trace of Algorithm OPT, it becomes apparent that there is no need to fall back on the pessimistic variant

of Figure 76(a) to exclude deadline violations. Rather, it suffices to adapt the online scheduler such that it becomes slightly more conservative. This leads us to the third DVS scheme which is defined by the following rule: *At each event arrival/completion compute* $s(t)$ *according to Equation (6.6). If* $s(t) \leq 0.85\,\mathrm{GHz}$ *use speed* $s(t)$*, otherwise use speed* s_{max}. The corresponding execution trace is shown in Figure 76(c). In this trace the CPU switches to s_{max} already at $t = 7\,\mathrm{ms}$ which permits to fulfill both timing and speed constraints. The resulting energy consumption amounts to 10.92 mJ, which is 21% less compared to the offline DVS scheduler.

6.4.2.2 Definition of the Adaptive DVS Scheme

To achieve energy savings while satisfying the timing constraints, we propose the adaptive DVS scheduling scheme shown in Figure 77. At a scheduling point at time t, the scheme derives the speed $s(t)$ by applying Algorithm OPT. If $s(t)$ is less than or equal to a speed threshold s^*, it greedily executes at speed $s(t)$. Otherwise, it executes at speed $s_{\max}$ for resolving the burst of event arrivals. We say that the adaptive DVS scheme is *feasible* with a threshold speed s^*, if schedulability is guaranteed for all input event traces conforming to the arrival curve $\bar{\alpha}$. To maximize the energy savings, we require that the DVS scheme is as optimistic as possible. In other words, the objective of our method is to derive the *maximal* threshold speed s^*, such that the adaptive DVS scheme is feasible.

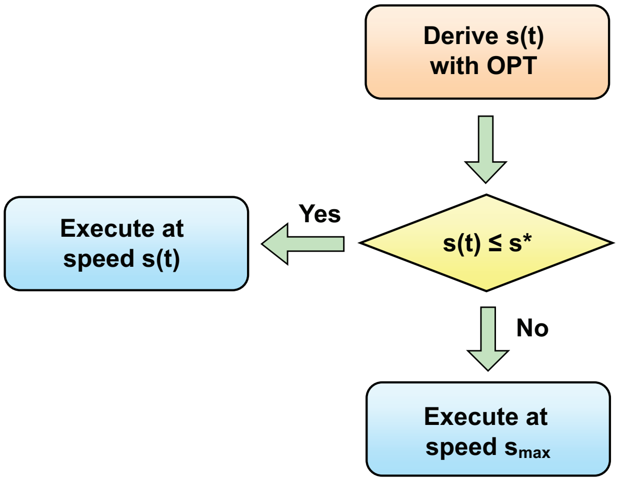


Fig. 77: Behaviour of adaptive DVS scheme at a generic scheduling point t

An important remark is that in the above adaptive scheme we do not employ the orignal OPT algorithm described in Section 6.2.2, but a slight variant of it. The original OPT algorithm is an *event-driven* scheduling algorithm, which means that the instant an event arrives, or its processing completes are scheduling points at which the speed is recomputed. In contrast, we employ a *time-driven* variant of Algorithm OPT in which the scheduling points are artificial clock ticks. The reason for this design choice is that we want to employ state-based models to derive s^* and verify the correctness of the scheme. More specifically, the problem is the encoding of the speed computation of Equation (6.6) in a state-based model. Equation (6.6) requires precise knowledge of the amount of remaining processing demand at a given time t. The same applies for the times left until the deadlines of the various input events. In a state-based model these quantities can only be correctly taken into account if the corresponding modelling formalism has a notion of continuous time, and if it supports computations on time variables. In our case, we want to employ the formalism of Timed Automata to model the system. The corresponding model checker Uppaal does, however, not support computations on clocks. This means that event-driven scheduling policies that are based on elapsed/remaining time such as OPT or EDF can not be precisely modelled with TA and Uppaal. Moreover, it is also not trivial to come up with a conservative TA approximation of the original OPT algorithm. The reason is that, on the one hand, any approximation that overestimates the actual speed $s(t)$ selected by OPT at time t is not safe, as it may ignore deadline violations, and on the other hand, any approximation that underestimates the actual speed $s(t)$ is not safe, as it might result in premature changes to $s_{\max}$ in the model, and hence again jeopardize the verification of deadlines.

Therefore, in order to avoid ambiguous results for the verification of a system, we restrict the adaptive DVS scheme to *time-driven* scheduling, a scenario that can be precisely captured in TA. We discretize time in the algorithm by introducing artificial clock ticks with period T. These ticks are counted in order to keep track of elapsed/remaining computation times. An event that arrives between two clock ticks is buffered, and affects the system only at the following tick. This means that an event that arrives at time t' will be released to the scheduler at time $\left\lceil \frac{t'}{T} \right\rceil T$. Similarly, we anticipate the deadline of the event from $t' + D$ to $\left\lfloor \frac{t'+D}{T} \right\rfloor T$. The resulting algorithm is an adaptive and time-driven variant of the original OPT algorithm.

6.4.3 Parametrization and Verification

The key issue for the design of the adaptive DVS scheduler described above is to determine the maximal threshold speed s^*. In this section, we introduce a method that permits us to derive this parameter for a given non-deterministic input event stream. The method exploits the formalism of TA to represent the adaptive DVS scheme, and to verify its correctness. The choice of a TA model is motivated by the fact that the behaviour of the adaptive scheduler is clearly state-dependent. In fact, the scheduler selects the current execution speed based on the actual history of event arrivals. Hence, to model the scheduler, traditional stateless MPA components are not helpful. As shown in [CST09], it is also not trivial to employ MPA to analyze the maximum speed that Algorithm OPT requires.

The proposed design method uses an arrival curve to abstractly describe the input event stream. To interface the event stream model with the TA based model of the DVS scheduler, the method employs the hybrid analysis framework introduced in Chapter 5. Figure 78 gives an overview of the proposed design method. It consists of the following components:

1. ***Simplification of arrival curve***
   To reduce the complexity of the system verification, we conservatively approximate the input arrival curve $\bar{\alpha}$ with a staircase curve $\bar{\alpha}'$ with non-decreasing step widths.

2. ***Event generator***
   We automatically derive a network of TA, which produces event traces that are bounded by the simplified staircase curve $\bar{\alpha}'$. This event generator is fully equivalent to $\bar{\alpha}'$ in the sense that it is able to produce *all* event traces that are constrained by $\bar{\alpha}'$.

3. ***TA based model of adaptive DVS processor***
   We use a TA based model of the adaptive DVS scheduler in which we employ clock discretization to keep track of the completed/remaing processing demands of events. The interaction of the event generator with the state-based DVS scheduler is modelled by means of a shared variable which represents the number of buffered input events. We employ the timed model checker Uppaal to derive the maximum threshold speed s^*. Specifically, we adopt a binary search strategy in which the timing constraints are verified for different values of s^*. If for a given value of s^* Uppaal reports a possible deadline violation, we have to be more conservative, that is, we reduce s^*. In contrast, if Uppaal confirms that for a given s^* all events meet their deadline, we try to use a more optimistic speed threshold, that is, we increase s^*. This verification procedure is repeated until the maximal threshold speed is approximated with enough precision.

An important observation is that the verification framework described above, which is based on (expensive) model checking of TA models, is

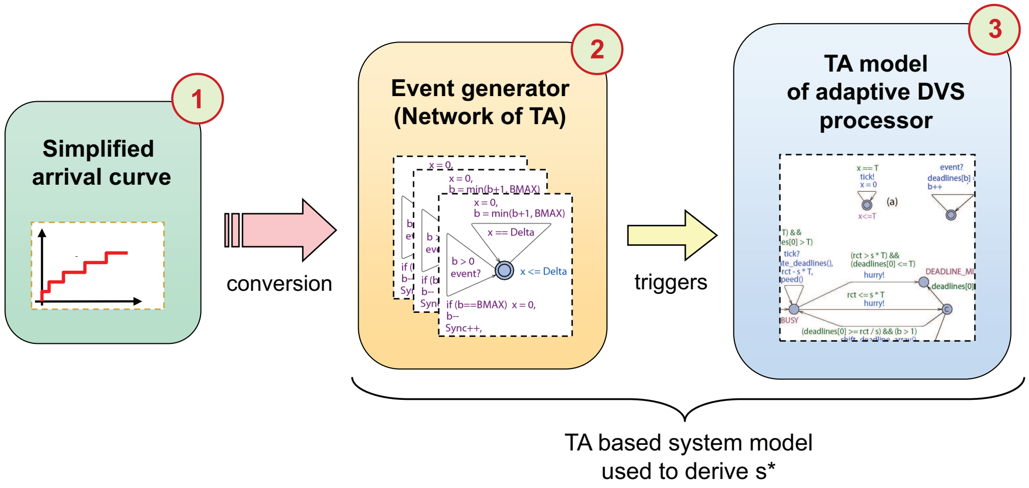

1
Simplified arrival curve
conversion
2
Event generator (Network of TA)
triggers
3
TA model of adaptive DVS processor
TA based system model used to derive s*

Fig. 78: Design framework for adaptive DVS scheme

used at design time only. It is employed offline to parameterize and verify the adaptive DVS scheduler for a given set of input event traces. However, there is no additional run-time overhead for the online DVS algorithm itself, which is just a simple adaptive variant of Algorithm OPT.

In the following, we describe the individual parts of the verification framework in more detail.

6.4.3.1 Conservative Simplification of the Input Arrival Curve

As described in Chapter 5, a general upper arrival curve $\bar{\alpha}$ can be conservatively approximated by a pseudo-concave arrival curve. Specifically, we approximate $\bar{\alpha}$ with a curve $\bar{\alpha}'$ of the form

$$\begin{aligned} \bar{\alpha}'(\Delta) &:= \min_i \left\{ \bar{\alpha}'_i(\Delta) \right\} \text{ with} \\ \bar{\alpha}'_i(\Delta) &:= N_i + \left\lfloor \frac{\Delta}{\delta_i} \right\rfloor . \end{aligned} \tag{6.10}$$

The curve $\bar{\alpha}'$ is a staircase curve with non-decreasing step widths which results from the minimum composition of several linear staircase functions $\bar{\alpha}'_i$, each specified by a parameter tuple (N_i, δ_i). The number of linear staircase functions $\bar{\alpha}'_i$ that are employed for representing the input event stream determines the precision of the curve approximation and affects the complexity of the TA based system verification.

6.4.3.2 TA Based Representation of the Arrival Curve

To translate the simplified arrival curve $\bar{\alpha}'$ to a network of cooperating TA, we employ the conversion mechanism described in Section 5.4.2.1. Specifically, we convert $\bar{\alpha}'$ to an event generator in the form of TA. The event generator consists of several UTA and an event scheduler (cf. Figure 56). We do not employ any LTA, as we represent an upper arrival curve only.

In Chapter 5, we have seen that this TA based model of arrival curves is very general in the sense that it can employ an arbitrary number of linear staircase components. However, there are also other more specific TA models to represent particular types of event streams. For instance, in [HV06], a TA model for periodic event streams with jitter is presented. While such dedicated models have a clearly limited application scope, they are at times more efficient in terms of verification compared to the above modelling principle.

6.4.3.3 TA Model and Verification of the Adaptive DVS Scheduler

In this section we introduce a TA model for the adaptive DVS scheduler described above. The model is based on a time-driven and discretized variant of Algorithm OPT.

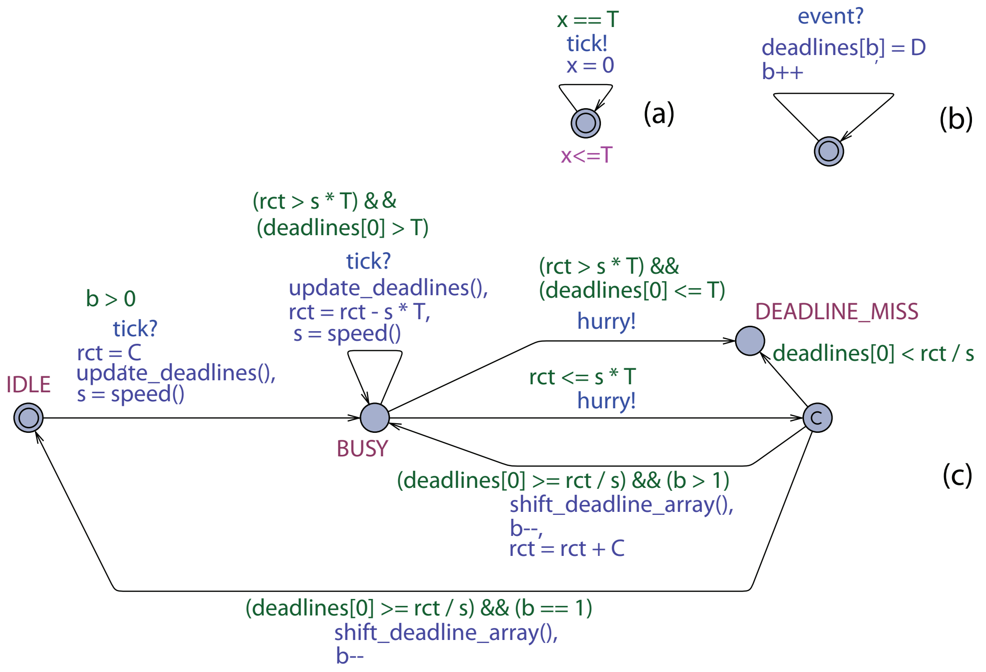


Fig. 79: TA model of discretized adaptive DVS scheduler

The TA model for a CPU that processes a single input event stream and implements the proposed adaptive DVS scheme is shown in Figure 79. The figure depicts a network of TA that cooperate by means of channels and global variables. The model assumes that all the events of the input stream have a constant execution demand C, expressed in processing units, and a constant relative deadline D, expressed in time units, where both C and D are integers. The model relies on an explicit representation of discrete time which is advanced by periodic clock ticks. Specifically, Automaton (a) broadcasts a signal *tick* every T time units (clock period). Automaton (b) handles the arrival of new events, by synchronizing with the event generator over the broadcast signal *event*. Every time a new event arrives, the automaton increments the counter variable b which represents the current number of buffered events. Note that the event arrivals are independent of the discrete clock ticks, meaning that new events can arrive at any point in time. However, in the proposed model an event arrival does not affect the CPU until the next clock tick, when the automaton either moves to the *BUSY* location if the CPU was idle, or determines a new processing speed if it was busy. The processing speed is represented by a global integer variable s. The value of s represents the number of processing units that the CPU provides per time unit. The model uses the integer array *deadlines* to keep track of the number of time units remaining until the deadlines of the individual events. At an event arrival, the corresponding position in the array is initialized with D, see Figure 79(b). At every clock tick, all elements in the deadline array are decreased by T. Note that at a clock tick, the deadline array is updated *before* the new processing speed is computed. This makes sure that the selected speed guarantees the schedulability of an event, even if it has been released with some delay (at the next clock tick following its actual arrival time).

The automaton of Figure 79(c) models the CPU itself. It manipulates a global integer variable *rct* that represents the remaining execution demand of the currently processed event. At each clock tick *rct* is decreased according to the selected processing speed. If the deadline of an event occurs within the next clock tick, but the processing of the event cannot be finished in time, the automaton immediately reports a deadline violation by moving to an appropriate state.[4] On the other hand, if the remaining execution demand for an event is less or equal than the processing service provided in a clock period, the location *BUSY* is immediately left, and one of the following three cases applies:

(1) The number of time units left to the deadline is not sufficient to complete the processing of the event. In this case, a deadline violation is

---

[4] An immediate reaction is guaranteed by the urgent channel *hurry* which is always ready to synchronize.

reported.

(2) The completion of the event processing happens in time, and there are no other events in the queue. In this case the transition to the location *IDLE* is taken.

(3) The completion of the event processing happens in time and there are other events in the queue. In this case the direct transition to the location *BUSY* is taken.

In both cases (2) and (3) the elements of the array *deadlines* are shifted by one position and the variable b is decremented.

Algorithm 6 `Compute speed`

```
function SPEED
    $int\ s \leftarrow 0,\ \ int\ s'$
    for $i \leftarrow 0, b-1$ do
        $s' = (rct + i * C)/deadlines[i]$
        $s \leftarrow max(s, s')$
    end for
    if $s > s^*$ then
        $s \leftarrow s_{max}$
    end if
    return $s$
end function
```

The function *speed* adopted in the model to compute the new processing speed is shown in Algorithm 6. The algorithm implements the described adaptive scheduling scheme by setting the processing speed to s_{max} if a speed above the threshold s^* is requested.

We employ the model checker Uppaal to verify whether an event stream can be correctly executed by the adaptive scheduler with a given threshold speed s^*. Specifically, we verify whether the following proposition holds for the TA model:

```
A[] (not CPU.DEADLINE_MISS)
```

By performing a simple binary search on s^*, we determine the *maximal* threshold speed that guarantees all timing constraints.

Finally, we would like to highlight that the discrete time representation of the scheduling scheme can be made arbitrarily precise by reducing the length of the clock period T. However, a more fine-grained time representation results in longer verification times. Hence, the system designer can balance accuracy and verification time of the model.

6.4.3.4 Generalization to Multiple Input Event Streams

The described approach is not limited to a single input event stream. In this section, we briefly sketch how the TA model introduced above can be adapted for the case of multiple input event streams.

We consider a CPU that processes the events of n input streams according to the Earliest Deadline First (EDF) scheduling policy. Each event is characterized by an execution demand C_i and a relative deadline D_i, $i \in n$. The extended TA model still uses a single counter b and a single array *deadlines* to handle all the input events, now originating from different streams. However, in a system with multiple input streams, the last arrived event does not necessarily have the farthest absolute deadline. Hence, we have to explicitly keep the elements in the array *deadlines* sorted, meaning that new deadline counters have to be inserted at the right position in the array. Moreover, under EDF scheduling, it can happen that the processing of an event is preempted by a newly arrived event with an earlier deadline. In our discrete time representation of the CPU, this case can be conservatively approximated by selecting the next event to be processed at each clock tick. The preemptions also imply that we have to explicitly store the remaining execution demand for each buffered event. This can be done by replacing the variable *rct* by an appropriate array of integers.

In principle, these adaptations extend the scope of the design method to systems with an arbitrary number of input streams. However, it is clear that the complexity of the verification increases drastically with the number of considered input streams. In other words, we expect that state space explosion is quickly encountered when applying the method to systems with multiple input streams.

6.4.4 Experimental Evaluation

In this section, we evaluate the performance of the proposed DVS scheduler in terms of energy consumption. Specifically, we compare the worst-case energy consumed by Algorithm SD, Algorithm OPT and the adaptive DVS scheme (denoted Algorithm AD) to process different event streams.

6.4.4.1 Experimental Setup

To compare the performance of the three algorithms, we use a set of six different input event streams adapted from [WT06b]. The considered streams are periodic event streams with jitter that are specified by the following parameters: period p, jitter j, minimum event inter-arrival time d, worst-case execution time C (at $s_{\max}$), and relative deadline D. Table 13 summarizes the parameter values for the six event streams.

	I	II	III	IV	V	VI
p	198	102	283	239	148	114
j	387	70	269	222	91	13
d	48	45	58	65	78	0
C	30	35	77	69	53	52
D	110	140	310	280	200	120

Tab. 13: Parameters for the six input event streams in [ms].

The streams have been selected such that if they are scheduled with Algorithm OPT, the scheduler violates the maximum system speed, $s_{\max} = 0.5\,\text{GHz}$. In other words, Algorithm OPT cannot be used for guaranteeing the schedulability of these streams on the considered CPU. The maximum speeds required by Algorithm OPT are shown in the second row of Table 14. These maximum speeds are computed with the method proposed in [CST09]. Specifically, we use approximative traces of length $3 \cdot D$ ms (where D is the relative deadline of a stream), which in [CST09] was shown to be a tight approximation. The first row of Table 14 shows the speeds required by Algorithm SD. These speeds are calculated with Equation 6.5.

	I	II	III	IV	V	VI
s_{SD}	0.44	0.38	0.42	0.4	0.39	0.47
s_{OPT}^{max}	0.513	0.505	0.501	0.506	0.506	0.506
s^* (T=2ms)	0.38	0.36	0.29	0.39	0.37	0.39

Tab. 14: Maximum speeds of Algorithms SD and OPT (first two rows). Threshold speeds s^* for Algorithm AD, determined assuming $s_{max} = 0.5$ (last row). All speeds are expressed in GHz.

For Algorithm AD, we derive the speed thresholds s^* with the model checker Uppaal based on the TA models of Section 6.4.3. In the TA model of the DVS scheduler, we employ discrete clock ticks with a granularity of $T = 2\,\text{ms}$. To improve the efficiency of the verification, we employ the dedicated TA event generators of [HV06] to represent the input event streams. This is possible due to the periodic nature of the considered streams. The resulting threshold speeds are shown in the last row of Table 14. Table 15 reports the run-times of Uppaal to compute these speed thresholds on a 64-bit Sun Fire X2200 M2 with 8GB RAM.

To evaluate the energy-efficiency of the different algorithms, we first use the RTC Toolbox to produce 10 random event traces for each of the streams. These traces are generated such that they are as close as possible

	I	II	III	IV	V	VI
T=2ms	210	262	16679	2973	459	2

Tab. 15: Run-times [s] for computing the threshold speeds s^* with discretization $T = 2$ ms.

to the arrival curve, each with a length of 20000 ms. We then simulate the execution of these traces by means of a simple discrete event simulator that implements the described scheduling schemes and monitors the energy consumption of the system. The simulator employs the power function $P(s) = 0.04 + \hbar \cdot 1.56(\frac{s}{0.5GHz})^3$ Watt, which is an approximation for the power dissipation of the Intel XScale architecture. Finally, for each algorithm we calculate the average energy consumed for scheduling the different traces of a particular stream.

6.4.4.2 Results

Figure 80 shows the average energy consumed by the algorithms to process the traces of the individual event streams. Since Algorithm OPT cannot guarantee the schedulability of the streams with a maximum speed of 0.5 GHz, its energy consumption is shown just as reference to illustrate the energy overhead of Algorithm AD. The chart shows that Algorithm AD is not much worse than Algorithm OPT in terms of consumed energy, on average 10%. On the other hand, for five of the streams, the adaptive DVS scheduler performs better than the static DVS scheduler, with an average energy saving of 22%.

There is, however, also one case (event stream VI), where Algorithm SD performs better than the adaptive approach. The reason for this result is that stream VI is almost fully periodic, that is, it exhibits only a very small amount of non-determinism. Therefore, executing at a constant speed is more energy-efficient than adapting the execution speed.

6.5 Related Work

The design of energy-efficient real-time computing systems has been an active research topic for many years. As a result, there is a large body of work on DVS and DPM scheduling for real-time systems. The existing approaches can be classified roughly into offline and online techniques. Offline algorithms as found in [YDS95, QH07, YK03] take scheduling decisions statically according to the expected worst-case workload of the system. Online algorithms as presented in [AMMMA01, MHQ05] decide on the task scheduling dynamically by adapting to the actual workload of the system. While online approaches can help to considerably reduce the

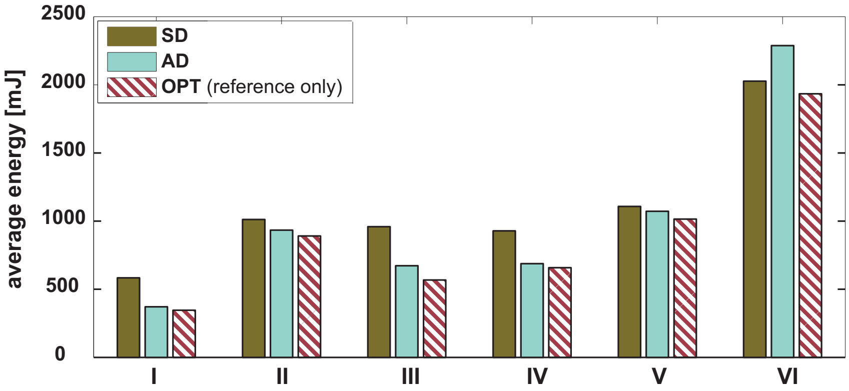


Fig. 80: Average energy consumed by the different algorithms for processing the event traces of the individual streams.

energy consumption in the case of light workloads, they often perform worse than offline methods for heavy workloads [YDS95]. Moreover, depending on the processed event stream, it can happen that online methods require an execution speed which is higher than the maximum available system speed. Therefore, the applicability of online methods has to be carefully verified a priori, which is the problem studied in [CST09].

Most of the studies on DVS scheduling in real-time systems assume that the input event streams are scheduled according to the Earliest Deadline First (EDF) policy, e.g., [YDS95, AMMMA01, KK05]. However, there are also several results for static priority scheduling. For instance, the authors of [YK03] propose an offline method to compute a near-optimal DVS schedule, assuming full a priori knowledge of event arrival times. In [QH07] a more efficient solution to the same problem is provided. Other approaches for static priority DVS scheduling assume periodic/sporadic event arrivals, e.g, [SR03, MHQ05]. In [SR03] the effect of discrete execution speeds on the energy performance of DVS scheduling is studied, and four different DVS schemes are presented. One of the four heuristics relies on priority-monotonic speed assignments, which is also the central idea of the offline method that we presented in Section 6.3. The authors of [MHQ05] take into account transition overheads for speed changes, while in [HJ08] energy-efficient preemption thresholds are discussed.

Almost all methods for static priority DVS scheduling are based on deterministic event traces, meaning that they either assume a priori knowledge of event arrival times, or presume periodic (or at most sporadic) event traces. Two exceptions are the methods proposed in [SK06] and [RHE$^+$06]. The authors of [SK06] consider mixed event streams (periodic and aperiodic event streams), and use scheduling servers to handle nondeterministic aperiodic event arrivals. They discuss interesting tradeoffs between energy consumption and response times for aperiodic events, but, cannot guarantee the deadlines of aperiodic events. In [RHE+06] the authors admit limited non-determinism for event arrival times, and express it by means of a PJD event model. They provide a stochastic method for power optimization based on multi-dimensional evolutionary algorithms.

The energy-aware processing of events with non-deterministic release times bounded by arrival curves has been recently explored in [MCT05, CST09]. For systems processing a single input event stream, the authors of [MCT05] explore how to choose the minimal constant execution speed that avoids an overflow of the input buffer. To the best of our knowledge, our work is the first contribution to DVS scheduling of *multiple* event streams with non-deterministic arrival times.

The problem of assigning static priorities to tasks executed by a preemptive scheduler has been widely studied. For periodic/sporadic tasks, optimal results are available with respect to schedulability, see rate-monotonic (RM) scheduling [LL73] and deadline-monotonic (DM) scheduling [LW82]. RM and DM cease to be optimal in the case of asynchronous task releases, that is, if the tasks have arbitrary offsets. The author of [Aud91] provides an optimal priority assignment method for this more general case. Also, several other extensions have been worked out for issues such as execution blocking [BA06], task re-executions [LB03], and limited priority levels [BYJ03]. All approaches for static priority assignment that we have found assume strictly periodic task activations, except for the initial offsets. This is not the case for [DB95], where the authors determine the optimal priorities for arbitrary aperiodic task activations, but again without schedulability guarantees. Moreover, all existing approaches optimize task priorities with respect to schedulability. Besides our work, we are not aware of any other methods for real-time systems that optimize task priorities with respect to energy consumption.

6.6 Summary

In this chapter, we studied the design and analysis of energy-efficient real-time systems that process non-deterministic input event streams, modelled by arrival curves. We proposed novel energy-aware design methods that rely on Dynamic Voltage Scaling (DVS) and employ MPA or TA to guarantee the timing constraints. Our methods are not bound to particular event stream models, but support arbitrary event arrival patterns, which considerably extends the application scope of DVS. Specifically, we considered two distinct design problems, namely the design of offline and online DVS schedulers.

In the first part of the chapter, we focussed on the offline DVS scheduling of multiple input event streams with static priorities. We introduced different algorithms that statically assign energy-efficient priorities and/or execution speeds to individual event streams. We considered both continuous and discrete execution speeds, and commented on the time complexity of the different approaches. The priority assignment algorithm is based on the observation that rearranging high-priority tasks does not affect the execution of a low-priority task. For the speed assignment algorithm, the main idea is to use priority-monotonic speeds in order to reduce the energy consumption. By interleaving the two algorithms, we obtained a heuristic that assures an energy-efficient processing of the input events, as well as the observation of all timing constraints. We evaluated the energy performance of the proposed offline algorithms by analyzing several test systems.

In the second part of the chapter, we concentrated on the online DVS scheduling of a single input event stream. Specifically, we explored an adaptive DVS scheme that dynamically switches between pessimistic and optimistic DVS scheduling. The adaptive strategy combines the advantages of both methodologies. For reducing the energy consumption, the adaptive scheme is as optimistic as possible when the workload is light. However, when the workload is heavy, the scheme switches to a pessimistic mode which guarantees the observation of deadlines. The change between the two modes is triggered when the execution speed overruns or underruns a threshold. To determine the most energy-efficient threshold speed, we introduced a modelling and verification framework based on Timed Automata. The framework employs the hybrid modelling technique of Chapter 5 to interface the MPA based description of the input event stream with the TA model of the DVS processor. By means of some simple experiments, we confirmed the beneficial properties of the adaptive DVS scheduler.

The proposed adaptive DVS approach can be easily extended to systems with discrete speeds, without falling back to voltage hopping. Since the restriction to discrete speeds reduces the search space, lower verification times are expected for this case. The adaptive approach is also not necessarily limited to a single input event stream. However, we expect a severe growth of the state space for systems with multiple input streams. Finally, note that executing at speed $s_{\max}$ is not necessarily the best option for the pessimistic mode. How to determine the most energy-efficient speed for the pessimistic mode is an open question.

7

Conclusions

This chapter summarizes the main results of this thesis and discusses potential directions of future research.

7.1 Main Results

In this thesis, we addressed the compositional performance evaluation of distributed embedded real-time systems. We introduced several formal methods for predicting the timing, memory, and energy performance of a system in early design stages. Our results build on top of an existing analytical method for performance evaluation, the MPA framework. The principal aims of this thesis are to extend the modelling scope, and increase the accuracy of current methods for compositional worst-case performance evaluation. We pointed out various situations in which current analytical techniques suffer abstraction losses, and also introduced formal methods for mitigating or eliminating such losses. More specifically, in this thesis we presented the following main results:

- We extended the modelling scope of MPA to systems with non-functional cyclic dependencies. We proved the correctness and convergence of fixpoint iterations in MPA, and showed how to obtain valid initial approximations. These results were achieved by developing a general operational semantic underlying the abstraction of MPA.

- We introduced a technique to represent the merging and splitting of event streams in MPA. The method is based on the concept of

Event Count Curves, an abstraction to characterize structured event streams. The new modelling technique seamlessly integrates into MPA and extends its modelling scope. As a result, abstraction losses are considerably reduced for systems with join/fork scenarios.

- We presented a novel hybrid methodology for the performance evaluation of distributed real-time systems, which combines analytical and state-based system evaluation. In the resulting framework, system components can be abstracted by either MPA or Timed Automata. The hybrid approach combines the benefits of both domains. On one hand, it drastically reduces the abstraction losses experienced with MPA due to coarse component models, and on the other, it limits the state-space explosion problem to the level of single system components.
- We showed how MPA can be applied to the design of energy-efficient real-time systems. In particular, we presented two novel design methods for offline and online DVS scheduling of real-time event streams. In contrast to other existing solutions, our methods cope with non-deterministic input streams, which considerably extends the application scope of DVS.

7.2 Outlook

The work presented in this thesis substantially augments the modelling scope and analysis accuracy of the MPA framework. Nevertheless, there still exists potential for further extensions and improvements. The following list identifies several possible directions for future research.

- ***Timing correlations***
  Arrival and service curves are a powerful instrument to model bounded non-determinism in streams. However, the interval-based representations have also a clear drawback: They cannot capture any timing correlations between streams which can often be observed in distributed systems. For instance, in MPA there is no means to represent relative phase shifts, or any other correlation between event traces in the time domain. As a consequence, severe abstraction losses may be experienced when analyzing systems that exhibit such kinds of correlations. The reason is that the correlations often foreclose the existence of worst-case situations which are considered by the MPA model. How to refine MPA models such that they capture and exploit knowledge about timing correlations is a challenging open question.

- ***Accurate Analysis of Cyclic Systems***
  In Chapter 3 we showed how systems with cyclic dependencies can be safely analyzed in MPA by means of fixpoint iterations. However, we also observed that modular methods such as MPA or SymTA/S often yield overly pessimistic bounds in the presence of cyclic dependencies. An approach to effectively prevent or mitigate abstraction losses in fixpoint iterations would be highly beneficial.

- ***Scheduling Servers***
  Scheduling servers such as the Deferrable Server, the Total Bandwidth Server, the Constant Bandwidth Server etc. are a powerful tool to contain the influence of a particular input stream on the schedulability of other streams. For instance, a scheduling server can be used for processing an aperiodic and potentially unbounded input stream in a best-effort fashion next to a set of periodic streams with hard timing requirements. Similarly, scheduling servers may be employed to isolate the effects of faulty input streams on a system. In future research, the modelling scope of MPA could be extended by designing abstract performance components for scheduling servers. In particular, it would be useful to extend current approaches for timing analysis of servers to the case of arbitrary input streams bounded by arrival curves. Another promising research direction is the design of new servers with particular guarantees, e.g., workload conservation.

- ***Sensitivity Analysis and Robustness***
  Apart from computing the performance metrics of a system at a specific design point, it is often also important to conduct a sensitivity analysis for the performance of the system. Specifically, it is significant to understand what effect (small) variations of system parameters have on the performance of the system. In future work, one could develop techniques for performing sensitivity analysis within the framework of MPA. The major application scenario would be the design of robust systems, that is, systems which exhibit insignificant sensitivity with respect to interferences and variations in streams.

- ***Coupling with other formalisms***
  In Chapter 5 we combined MPA with the formalism of Timed Automata. This coupling appeared most natural due to the implicit notion of time in TA. However, there are also other options to integrate MPA with formal verification. For instance, we can think of a hybrid analysis framework that combines analytical performance evaluation with sat-solving. The interfaces between these domains

could potentially borrow from the conversion patterns described in this thesis.

Bibliography

[ABS06] K. Albers, F. Bodmann, and F. Slomka. Hierarchical Event Streams and Event Dependency Graphs: A new Computational Model for Embedded Real-Time Systems. *18th Euromicro Conference on Real-Time Systems*, pages 97–106, 2006.

[AD94] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AFM+] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES: ATool for Schedulability Analysis and Code Generation of Real-Time Systems. `http://www.timestool.com/`.

[ALE02] T. M. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, 2002.

[AM10] K. Altisen and M. Moy. Arrival Curves for Real-Time Calculus: the Causality Problem and its Solutions. In J. Esparza and R. Majumdar, editors, *TACAS*, pages 358–372, March 2010.

[AMMMA01] H. Aydin, R. Melhem, D. Mossé, and P. Mejía-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *IEEE Real-Time Systems Symposium*, pages 95–105, 2001.

[Aud91] N. C. Audsley. Optimal Priority Assignment And Feasibility Of Static Priority Tasks With Arbitrary Start Times. Technical Report YCS 164, Department of Computer Science, University of York, 1991.

[BA06] K. Bletsas and N. Audsley. Optimal Priority Assignment in the Presence of Blocking. *Information Processing Letters*, 99(3):83–86, 2006.

[BBMS10] S. Baruah, V. Bonifaci, A. Marchetti-Spaccamela, and S. Stiller. Improved Multiprocessor Global Schedulability Analysis. *Real-Time Systems*, 46(1):3–24, 2010.

[BDM+] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS: A Model-Checking Tool for Real-Time Systems.
`http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/`.

[BHM08] A. W. Brekling, M. R. Hansen, and J. Madsen. Models and formal verification of multiprocessor system-on-chips. *The Journal of Logic and Algebraic Programming*, 77(1-2):1–19, 2008.

[Bip] The BIP framework and tool.
http://www-verimag.imag.fr/BIP,196.html.
Verimag Research Center, Gieres, France.

[BK08] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, 2008.

[BKP04] N. Bansal, T. Kimbrel, and K. Pruhs. Dynamic Speed Scaling to Manage Energy and Temperature. In *Proceedings of the Symposium on Foundations of Computer Science*, pages 520–529, 2004.

[BLL+] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL—A Tool Suite for Automatic Verification of Real-Time Systems.
`http://www.uppaal.com/`.

[BLL+96] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL—A Tool Suite for Automatic Verification of Real-Time Systems. In *Proceedings of the DIMACS/SYCON Workshop on Hybrid Systems III: Verification and Control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[BP05] N. Bansal and K. Pruhs. Speed Scaling to Manage Temperature. In *STACS*, pages 460–471, 2005.

[BPN+04] A. Bobrek, J. J. Pieper, J. E. Nelson, J. M. Paul, and D. E. Thomas. Modeling Shared Resource Contention Using a Hybrid Simulation/Analytical Approach. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, page 21144, Washington, DC, USA, 2004. IEEE Computer Society.

[But97] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, Boston, 1997.

[BY04] J. Bengtsson and W. Yi. Timed Automata: Semantics, Algorithms and Tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2004.

[BYJ03] X. Bin, Y. Yang, and S. Jin. Optimal Fixed Priority Assignment with Limited Priority Levels. In *Advanced Parallel Programming Technologies, 5th International Workshop (APPT 2003)*, pages 194–203, 2003.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[Cha00] C. S. Chang. *Performance Guarantees in Communication Networks*. Springer-Verlag, London, UK, 2000.

[CHL+03] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.

[CKT03] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 10190–10195. IEEE Computer Society, March 2003.

[CLS+06] Samarjit Chakraborty, Yanhong Liu, Nikolay Stoimenov, Lothar Thiele, and Ernesto Wandeler. Interface-Based Rate Analysis of Embedded Systems. In *7th IEEE International Real-Time Systems Symposium (RTSS 06)*, pages 25–34, Rio de Janeiro, Brasil, 2006.

[CPT05] S. Chakraborty, L. T. X. Phan, and P. S. Thiagarajan. Event Count Automata: A State-Based Model for Stream Processing Systems. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 87–98, 2005.

[CSB92] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-Power CMOS Digital Design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, 1992.

[CST09] J.-J. Chen, N. Stoimenov, and L. Thiele. Feasibility Analysis of On-Line DVS Algorithms for Scheduling Arbitrary Event Streams. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 261–270, 2009.

[DB95] R. Davis and A. Burns. Optimal priority assignment for aperiodic tasks with firm deadlines in fixed priority preemptive systems. *Information Processing Letters*, 53(5):249–254, 1995.

[DMS09] H. Dierks, A. Metzner, and I. Stierand. Efficient Model-Checking for Real-Time Task Networks. In *2nd International Conference on Embedded Software and Systems*, pages 11–18. IEEE Computer Society, 2009.

[FPY02] E. Fersman, P. Pettersson, and W. Yi. Timed Automata with Asynchronous Processes: Schedulability and Decidability. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 67–82, London, UK, 2002. Springer-Verlag.

[GFB03] J. Goossens, S. Funk, and S. Baruah. Priority-Driven Scheduling of Periodic Task Systems on Multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[GGPD01] M. González Harbour, J. J. Gutiérrez García, J. C. Palencia Gutiérrez, and J. M. Drake Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. In *Proceedings of 13th Euromicro Conference on Real-Time Systems*, pages 125–134. IEEE Computer Society, 2001.

[GLMS02] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[GMM90] C. Ghezzi, D. Mandrioli, and A. Morzenti. TRIO: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.

[Ham08] A. Hamann. *Iterative Design Space Exploration and Robustness Optimization for Embedded Systems*. Cuvillier Verlag Publisher, Göttingen, Germany, 2008.

[HE05] R. Henia and R. Ernst. Context-Aware Scheduling Analysis of Distributed Systems with Tree-Shaped Task-Dependencies. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 480–485. IEEE Computer Society, 2005.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, page 278, Washington, DC, USA, 1996. IEEE Computer Society.

[HHJ$^+$05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System Level Performance Analysis - the SymTA/S Approach. *Computers and Digital Techniques, IEE Proceedings -*, 152(2):148–166, March 2005.

[HJ08] X. He and Y. Jia. Leakage-aware energy efficient scheduling for fixed-priority tasks with preemption thresholds. In *international conference on Advanced Data Mining and Applications*, pages 379–390, 2008.

[HKH$^+$09] W. Haid, M. Keller, K. Huang, I. Bacivarov, and L. Thiele. Generation and Calibration of Compositional Performance Analysis Models for Multi-Processor Systems. In *Proceedings of International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, pages 92–99, Samos, Greece, July 2009. IEEE Press.

[HT07] W. Haid and L. Thiele. Complex Task Activation Schemes in System Level Performance Analysis. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 173–178. ACM, 2007.

[HTSD07] K. Huang, L. Thiele, T. Stefanov, and E. Deprettere. Performance Analysis of Multimedia Applications using Correlated Streams. In *Design, Automation and Test in Europe (DATE 07)*, pages 912–917, Nice, France, April 2007.

[HV06] M. Hendriks and M. Verhoef. Timed Automata Based Analysis of Embedded System Architectures. In *Workshop on Parallel and Distributed Real-Time Systems (WPDRTS)*. IEEE, 2006.

[Jer05] M. Jersak. *Compositional Performance Analysis for Complex Embedded Applications*. PhD thesis, Technical University of Braunschweig, 2005.

[JHE04] M. Jersak, R. Henia, and R. Ernst. Context-Aware Performance Analysis for Efficient Embedded System Design. In *Design, Automation and Test in Europe Conference and Exposition (DATE)*, pages 1046–1051. IEEE Computer Society, 2004.

[JP86] M. Joseph and P. K. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[JPG04] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the Design Automation Conference*, pages 275–280, 2004.

[JPTY08] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi. Cyclic Dependencies in Modular Performance Analysis. In *Proceedings of the 8th ACM international conference on Embedded software (EMSOFT'08)*, pages 179–188, New York, NY, USA, October 2008. ACM.

[JRE05] M. Jersak, K. Richter, and R. Ernst. Performance Analysis for Complex Embedded Applications. *International Journal of Embedded Systems*, 1(1/2):33–49, 2005.

[Kah74] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *In Proceeding of the Congress on Information Processing 1974 (IFIP'74)*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[KHET07] S. Künzli, A. Hamann, R. Ernst, and L. Thiele. Combined approach to system level performance analysis of embedded systems. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 63–68. ACM, 2007.

[KK05] W.-C. Kwon and T. Kim. Optimal Voltage Allocation Techniques for Dynamically Variable Voltage Processors. *ACM Transactions on Embedded Computing Systems*, 4(1):211–230, 2005.

[KMY07] P. Krcal, L. Mokrushin, and W. Yi. A tool for compositional analysis of timed systems by abstraction (extended abstract). In *Proceedings of 19th Nordic Workshop on Programming Theory (NWPT07)*, October 2007.

[KPBT06] S. Künzli, F. Poletti, L. Benini, and L. Thiele. Combining Simulation and Formal Methods for System-Level Performance Analysis. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE)*, pages 236–241, 3001 Leuven, Belgium, Belgium, March 2006. European Design and Automation Association.

[LB03] G. M. Lima and A. Burns. An Optimal Fixed-Priority Assignment Algorithm for Supporting Fault-Tolerant Hard Real-Time Systems. *IEEE Transactions on Computers*, 52:1332–1346, 2003.

[Leh90] J.P. Lehoczky. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of 11th Real-Time Systems Symposium (RTSS)*, pages 201–209. IEEE Computer Society, December 1990.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM95] Y. S. Li and S. Malik. Performance Analysis of Embedded Software using Implicit Path Enumeration. *ACM SIGPLAN Notices*, 30(11):88–98, 1995.

[LRD04] K. Lahiri, A. Raghunathan, and S. Dey. Design Space Exploration for Optimizing On-Chip Communication Architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):952–961, June 2004.

[LS00] S. Lee and T. Sakurai. Run-time voltage hopping for low-power real-time systems. In *DAC*, pages 806–809. ACM, 2000.

[LT01] J. Y. Le Boudec and P. Thiran. *Network Calculus - A Theory of Deterministic Queuing Systems for the Internet*. Number 2050 in Lecture Notes in Computer Science (LNCS). Springer, 2001.

[LW82] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[Mak06] A. Maksyagin. *Modeling multimedia workloads for embedded system design*. Shaker Verlag, Aachen, Germany, 2006.

[MCT05] A. Maxiaguine, S. Chakraborty, and L. Thiele. DVS for Buffer-Constrained Architectures with Predictable QoS-Energy Tradeoffs. In *CODES+ISSS*, pages 111–116, 2005.

[Met] Metropolis: Design Environment for Heterogeneous Systems.
`http://embedded.eecs.berkeley.edu/metropolis/`.
Center for Electronic System Design, UC Berkeley, Berkeley, CA, USA.

[MHQ05] B. Mochocki, X. S. Hu, and G. Quan. Practical on-line dvs scheduling for fixed-priority real-time systems. In *RTAS*, pages 224–233, 2005.

[Mos] Modeling, Simulation, and Evaluation of Systems (MOSES). `http://www.tik.ee.ethz.ch/ moses/`. Computer Engineering and Networks Lab, Swiss Federal Institute of Technology (ETHZ), Zurich, Switzerland.

[NM09] G. Nicolescu and P. J. Mosterman. *Model-Based Design for Embedded Systems*. CRC Press (Taylor & Francis Group), 2009.

[ns2] ns-2 Network Simulator.
`http://www.isi.edu/nsnam/ns/`.

[NSE09] M. Negrean, S. Schliecker, and R. Ernst. Response-Time Analysis of Arbitrarily Activated Tasks in Multiprocessor Systems with Shared Resources. In *Design, Automation and Test in Europe (DATE)*, pages 524–529. IEEE, 2009.

[NuS] NuSMV: A new symbolic model checker.
`http://nusmv.fbk.eu/`.

[Ovp] Open Virtual Platforms by Imperas Software Ltd.
`http://www.ovpworld.org/`.

[PCT08] L.T.X. Phan, S. Chakraborty, and P.S. Thiagarajan. A Multi-mode Real-Time Calculus. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2008)*, pages 59–69. IEEE Computer Society, 2008.

[PCTT07] L. T. X. Phan, S. Chakraborty, P. S. Thiagarajan, and L. Thiele. Composing Functional and State-Based Performance Models for Analyzing Heterogeneous Real-Time Systems. In *Proceedings of the 28th IEEE Real-Time Systems Symposium (RTSS 2007)*, pages 343–352. IEEE Computer Society, 2007.

[PEP00] P. Pop, P. Eles, and Z. Peng. Performance Estimation for Embedded Systems with Data and Control Dependencies. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign*, pages 62–66. ACM Press, 2000.

[PEP02] T. Pop, P. Eles, and Z. Peng. Holistic Scheduling and Analysis of Mixed Time/Event-triggered Distributed Embedded Systems. In *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, pages 187–192. ACM Press, 2002.

[Per] S. Perathoner. PESIMDES Simulator. `http://www.mpa.ethz.ch/PESIMDES`.

[Per06] S. Perathoner. Evaluation and Comparison of Performance Analysis Methods for Distributed Embedded Systems. Master's thesis, ETH Zurich, 2006.

[PG98] J. C. Palencia Gutiérrez and M. González Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *Proceedings of the 19th Real-Time Systems Symposium (RTSS)*. IEEE Computer Society, 1998.

[PG03] J. C. Palencia Gutiérrez and M. González Harbour. Offset based Response Time Analysis of Distributed Systems Scheduled under EDF. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 3–12. IEEE Computer Society, 2003.

[Poo] Parallel Object-Oriented Specification Language (POOSL). `http://www.es.ele.tue.nl/she/index.php?select=32`. Electronic Systems Group, University of Technology Eindhoven, Eindhoven, The Netherlands.

[PTL] PTLsim by Strandera Corp. `http://www.ptlsim.org/`.

[Pto] The Ptolemy project. `http://ptolemy.eecs.berkeley.edu/`.

Center for Hybrid and Embedded Software Systems (CHESS), UC Berkeley, Berkeley, CA, USA.

[QH07] G. Quan and X. S. Hu. Energy efficient dvs schedule for fixed-priority real-time systems. *ACM Transactions on Embedded Computing Systems*, 6(4):29, 2007.

[QS82] J. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science (LNCS)*, pages 337–351. Springer Berlin / Heidelberg, 1982.

[Rac09] R. Racu. *Performence Characterziation and Sensitivity Analysis of Real-Time Embedded Systems*. Sierke Verlag Publisher, Göttingen, Germany, 2009.

[RE08] J. Rox and R. Ernst. Modeling Event Stream Hierarchies with Hierarchical Event Models. In *Design, Automation and Test in Europe (DATE)*, pages 492–497. IEEE, 2008.

[RHE$^+$06] R. Racu, A. Hamann, R. Ernst, B. Mochocki, and X. S. Hu. Methods for power optimization in distributed embedded systems with real-time requirements. In *CASES*, pages 379–388, 2006.

[Ric05] K. Richter. *Compositional Performance Analysis Using Standard Event Models - The SymTA/S Approach*. PhD thesis, Technical University of Braunschweig, 2005.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A formal approach to MpSoC performance verification. *Computer*, 36(4):60–67, April 2003.

[RKZ95] A. Raha, S. Kamat, and W. Zhao. Guaranteeing End-to-End Deadlines in ATM Networks. In *15th IEEE International Conference on Distributed Computing Systems (ICDCS'95)*, pages 60–68, 1995.

[RLH$^+$07] R. Racu, L. Li, R. Henia, A. Hamann, and R. Ernst. Improved Response Time Analysis of Tasks Scheduled under Preemptive Round-Robin. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS 2007)*, pages 179–184, 2007.

[SDI+08] S. Stein, J. Diemer, M. Ivers, S. Schliecker, and R. Ernst. On the Convergence of the SymTA/S Analysis. Technical report, TU Braunschweig, Braunschweig, Germany, November 2008.

[SE09] S. Schliecker and R. Ernst. A Recursive Approach to End-To-End Path Latency Computation in Heterogeneous Multiprocessor Systems. In *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 433–442. ACM, 2009.

[Sim] SimpleScalar LLC.
`http://www.simplescalar.com/`.

[SJDL05] H. Schiøler, J.J. Jessen, J. Dalsgaard, and K.G. Larsen. Network Calculus for Real Time Analysis of Embedded Systems with Cyclic Task Dependencies. In Gongzhu Hu, editor, *Proceedings of 20th International Conference on Computers and Their Applications, CATA 2005, March 16-18, 2005, Louisiana*, pages 326–332. ISCA, 2005.

[SK06] D. Shin and J. Kim. Dynamic voltage scaling of mixed task sets in priority-driven systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 25(3):438–453, 2006.

[Spi] The Spin Model Checker.
`http://spinroot.com`.

[SPLT10] Urban Suppiger, Simon Perathoner, Kai Lampka, and Lothar Thiele. Modular performance analysis of large-scale distributed embedded systems: An industrial case study. Technical Report 330, Computer Engineering and Networks Laboratory, ETH Zurich, 2010.

[SR03] S. Saewong and R. Rajkumar. Practical Voltage-Scaling for Fixed-Priority RT-Systems. In *RTAS*, page 106, 2003.

[SRIE08] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing Accurate Event Models for the Analysis of Heterogeneous Multiprocessor Systems. In *Proceedings of 6th International Conference on Hardware Software Codesign and System Synthesis (CODES-ISSS)*, Atlanta (GA), USA, October 2008.

[Sym] SymTA/S by Symtavision GmbH.
`http://www.symtavision.com/`.

[Sys] The Open SystemC Initiative. `http://www.systemc.org/`.

[TBW94] K. W. Tindell, A. Burns, and A. J. Wellings. An extendible approach for analyzing fixed priority hard real-time tasks. *Real-Time Systems*, 6(2):133–151, 1994.

[TC94] K. Tindell and J. Clark. Holistic Schedulability Analysis for Distributed Hard Real-Time Systems. *Microprocessing and Microprogramming - Euromicro Journal (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *Proceedings of International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104, 2000.

[TS] L. Thiele and N. Stoimenov. Real-Time Simulation (RTS) Toolbox. `http://www.mpa.ethz.ch/Rtstoolbox`.

[TS09] L. Thiele and N. Stoimenov. Modular Performance Analysis of Cyclic Dataflow Graphs. In *EMSOFT 09: Proceedings of the 9th ACM International Conference on Embedded Software*, pages 127–136, Grenoble, France, 2009.

[TT] TrueTime Simulator. `http://www.control.lth.se/truetime/`.

[TW] L. Thiele and E. Wandeler. Performance Analysis of Distributed Embedded Systems. In *Embedded Systems Handbook*. Richard Zurawski (Editor), CRC Press, 2005, 1st edition.

[Upp] UPPAAL—A Tool Suite for Automatic Verification of Real-Time Systems. `http://www.uppaal.org/`.

[Wan06] E. Wandeler. *Modular Performance Analysis and Interface-Based Design for Embedded Real-Time Systems*. Shaker Verlag, Aachen, Germany, 2006.

[WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and

P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[Wil] R. Wilhelm. Determining Bounds on Execution Times. In *Handbook on Embedded Systems*. R. Zurawski (Editor), CRC Press, 2005, 1st edition.

[WMT05] Ernesto Wandeler, Alexandre Maxiaguine, and Lothar Thiele. Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications. *Real-time Systems*, 29(2):205–225, 2005.

[WT] E. Wandeler and L. Thiele. Real-Time Calculus (RTC) Toolbox.
`http://www.mpa.ethz.ch/Rtctoolbox`.

[WT06a] E. Wandeler and L. Thiele. Interface-Based Design of Real-Time Systems with Hierarchical Scheduling. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243–252, Washington, DC, USA, 2006. IEEE Computer Society.

[WT06b] E. Wandeler and L. Thiele. Optimal TDMA Time Slot and Cycle Length Allocation for Hard Real-Time Systems. In *ASP-DAC*, pages 479–484, 2006.

[WTVL06] E. Wandeler, L. Thiele, M. Verhoef, and P. Lieverse. System Architecture Evaluation Using Modular Performance Analysis: A Case Study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667, 2006.

[XMM04] F. Xie, M. Martonosi, and S. Malik. Intraprogram Dynamic Voltage Scaling: Bounding Opportunities with Analytic Modeling. *ACM Trans. Archit. Code Optim.*, 1(3):323–367, 2004.

[Yat93] R. K. Yates. Networks of Real-Time Processes. In *CONCUR'1993*, pages 384–397, 1993.

[YDS95] F. Yao, A. Demers, and S. Shenker. A Scheduling Model for Reduced CPU Energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 374–382. IEEE, 1995.

[YK03] H.-S. Yun and J. Kim. On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *ACM Transactions on Embedded Computing Systems*, 2(3):393–430, 2003.

[You07] M. T. Yourst. PTLsim: A Cycle Accurate Full System x86-64 Microarchitectural Simulator. In *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34, April 2007.

[YW95] T. Y. Yen and W. Wolf. Performance Estimation for Real-Time Distributed Embedded Systems. In *Proceedings of the 1995 International Conference on Computer Design*, pages 64–71. IEEE Computer Society, 1995.

[ZMM04] D. Zhu, R. G. Melhem, and D. Mossé. The effects of energy management on reliability in real-time embedded systems. In *ICCAD*, pages 35–40, 2004.

A

Acronyms

CAN	Controller Area Network
CPU	Central Processing Unit
DM	Deadline Monotonic
DPM	Dynamic Power Management
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
ECA	Event Count Automata
ECC	Event Count Curve
EDF	Earliest Deadline First
FIFO	First In First Out
FP	Fixed Priority
GPC	Greedy Processing Component
HCS	Heterogeneous Communication System
I/O	Input/Output
MAST	Modeling and Analysis Suite for Real-Time Applications
MPA	Modular Performance Analysis
MPSoC	Multiprocessor System-on-Chip
NAC	Network Access Controller
PJD	Periodic with Jitter
RM	Rate Monotonic
RTC	Real-Time Calculus
SEM	Standard Event Model
SymTA/S	Symbolic Timing Analysis for Systems
TA	Timed Automata
TDMA	Time Division Multiple Access
WCRT	Worst-Case Response Time
WFQ	Weighted Fair Queuing

List of Publications

The following list summarizes the publications that constitute the basis of this thesis. The pertinent chapters of the thesis are indicated in brackets.

S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, M. González Harbour. **Influence of Different System Abstractions on the Performance Analysis of Distributed Real-Time Systems**. In *7th ACM & IEEE International Conference on Embedded Software (EMSOFT)*, October 2007. (Chapter 2)

B. Jonsson, S. Perathoner, L. Thiele, W. Yi. **Cyclic Dependencies in Modular Performance Analysis**. In *8th ACM International Conference on Embedded Software (EMSOFT)*, October 2008. (Chapter 3)

S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, M. González Harbour. **Influence of Different Abstractions on the Performance Analysis of Distributed Hard Real-Time Systems**. In *Design Automation for Embedded Systems*, Springer Science+Business Media, Vol. 13, No. 1-2, June 2009. (Chapter 2)

K. Lampka, S. Perathoner, L. Thiele. **Analytic Real-Time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-Time Systems**. In *9th ACM International Conference on Embedded Software (EMSOFT)*, October 2009, *ACM SigBed EMSOFT Best paper award*. (Chapter 5)

L. Thiele, S. Perathoner. **Performance Prediction of Distributed Platforms**. In *Model-Based Design of Heterogeneous Embedded Systems*, Gabriela Nicolescu and Pieter J Mosterman, Editors, CRC Press, November, 2009. (Chapter 2)

S. Perathoner, T. Rein, L. Thiele, K. Lampka, J. Rox. **Modeling Structured Event Streams in System Level Performance Analysis**. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*, April 2010. (Chapter 4)

S. Perathoner, J.-J. Chen, L. Thiele. **Energy-Efficient Static Priority and Speed Assignment for Real-Time Tasks with Non-Deterministic Release Times**. In *16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, August 2010.
(Chapter 6)

K. Lampka, S. Perathoner, L. Thiele. **Analytic Real-Time Analysis and Timed Automata: A Hybrid Methodology for the Performance Analysis of Embedded Real-Time Systems**. In *Design Automation for Embedded Systems*, Springer Science+Business Media, Vol. 14, No. 3, September 2010. (Chapter 5)

S. Perathoner, J.-J. Chen, K. Lampka, N. Stoimenov, L. Thiele. **Combining Optimistic and Pessimistic DVS Scheduling: An Adaptive Scheme and Analysis**. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2010. (Chapter 6)

It follows a list of publications that are not covered in this thesis.

N. Stoimenov, S. Perathoner, L. Thiele. **Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling**. In *Design, Automation and Test in Europe (DATE)*, April 2009.

S. Perathoner, K. Lampka, L. Thiele. **Composing Heterogeneous Components for System-wide Performance Analysis**. In *Design, Automation and Test in Europe (DATE)*, March 2011.

Curriculum Vitae

Name	Simon Perathoner
Date of Birth	12 June 1981
Citizen of	Italy

Education:

12/2006–04/2011	ETH Zurich, Computer Engineering and Networks Laboratory PhD studies under the supervision of Prof. Dr. L. Thiele
09/2004–10/2006	UNITECH International Scholar Graduation with UNITECH International Certificate
10/2003–04/2006	Politecnico di Milano, Italy Master studies in Computer Engineering 17 months visiting student at ETH Zurich, Switzerland Graduation as *dottore magistrale in ingegneria informatica* (equivalent to MSc. Computer Engineering)
10/2000–09/2003	Politecnico di Milano, Italy Bachelor studies in Computer Engineering Graduation as *dottore in ingegneria informatica* with distinction (equivalent to BSc. Computer Engineering)
09/1995–06/2000	Gewerbeoberschule Max Valier, Bolzano, Italy Secondary education with specialization in computer science Graduation with Matura and distinction

Professional Experience:

12/2006–04/2011	Research & Teaching Assistant at the Computer Engineering and Networks Laboratory of ETH Zurich
06/2006–09/2006	Internship as Embedded Software Engineer at Sony Corporation, Tokyo, Japan

Awards and Grants:

10/2009	ACM SigBed EMSOFT Best Paper Award
12/2004	Scholarship Noverino Faletti 2003 granted by Italian Association of Electronics, Computer and Communications Engineering